NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

INTEGRATION OF ASW HELICOPTER OPERATIONS AND ENVIRONMENT INTO NPSNET

by

Frederick Charles Lentz, III

September 1995

Thesis Advisor:	Michael J. Zyda
Co-Advisor:	John S. Falby

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE September 1995	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE INTEGRATION OF ASW HELICOPTER OPERATIONS AND ENVIRONMENT INTO NPSNET (U)	5. FUNDING NUMBERS
6. AUTHOR(S) Lentz, Frederick Charles, III	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Despite the increasing emphasis by the military on joint force operations, existing modelling and simulation programs, including NPSNET, fail to address joint operations and crew coordination. The problem is that previous work on NPSNET, the virtual environment and visual simulation platform developed by the Computer Science Department at the Naval Postgraduate School in Monterey, California, has focused primarily on individual ground force elements with little emphasis on naval forces or crew concepts. This restricts the practical use of the system to ground force training while ignoring joint force training with sea and air components and between crew members.

One solution to is expand the capability of NPSNET by incorporating a variety of vehicles from different components of the military with the added capability of multiple workstation control of a single vehicle. The approach taken is to expand NPSNET to simulate helicopter Anti-Submarine Warfare. This work focuses on realistic helicopter flight control, multiple workstation control of a single vehicle, and interface design between workstations controlling one vehicle.

NPSNET has become a more useful training tool for today's military forces by implementing more realistic helicopter flight controls and adding joint mission capabilities. The significance of this work is that a broad range of forces can receive valuable joint training and crew coordination training conducted in a virtual environment.

14. SUBJECT TERMS NPSNET, Interface, Helicopter, ASW, Network, Simulation, Virtual Environment.	15. NUMBER OF PAGES
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

INTEGRATION OF ASW HELICOPTER OPERATIONS AND ENVIRONMENT INTO NPSNET

Frederick Charles Lentz, III
Lieutenant, United States Navy
B.A., Virginia Tech, 1988

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1995

Author: ______________________________
Frederick Charles Lentz, III

Approved by: ______________________________
Michael J. Zyda, Thesis Advisor

____
John S. Falby, Thesis Co-Advisor

____
Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

Despite the increasing emphasis by the military on joint force operations, existing modelling and simulation programs, including NPSNET, fail to address joint operations and crew coordination. The problem is that previous work on NPSNET, the virtual environment and visual simulation platform developed by the Computer Science Department at the Naval Postgraduate School in Monterey, California, has focused primarily on individual ground force elements with little emphasis on naval forces or crew concepts. This restricts the practical use of the system to ground force training while ignoring joint force training with sea and air components and between crew members.

One solution to is expand the capability of NPSNET by incorporating a variety of vehicles from different components of the military with the added capability of multiple workstation control of a single vehicle. The approach taken is to expand NPSNET to simulate helicopter Anti-Submarine Warfare. This work focuses on realistic helicopter flight control, multiple workstation control of a single vehicle, and interface design between workstations controlling one vehicle.

NPSNET has become a more useful training tool for today's military forces by implementing more realistic helicopter flight controls and adding joint mission capabilities. The significance of this work is that a broad range of forces can receive valuable joint training and crew coordination training conducted in a virtual environment.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

This project is made possible through the tireless efforts of a number of people. Both directly and indirectly, their assistance was invaluable to completing this research. I would like to take this opportunity to thank those who provided any guidance or assistance to my efforts. Dr. Mike Zyda and Mr. John Falby provided me the opportunity to do research in something I found interesting, then gave me the guidance I needed to keep the research in focus and meaningful. Mr. Paul Barham was also instrumental to keeping this research going through the many logjams I encountered. His expertise in NPSNET was vital to getting and staying off the ground. Without them, this work would not have been possible.

I would also like to thanks everyone in the CS41 Graphics Track for their assistance, despite the fact that much of our work seemed to contradict each other at times. We were all able to peacefully coexist and come up with some good results. Special thanks to Al Shaffer for his programming expertise. He was instrumental in writing functioning source code.

In addition, I would like to thank Mr. Don Selvy at A&T, Inc. His efforts in helping me retrieve the helicopter control information I needed were vital to my understanding of how to make this helicopter fly.

Finally, my deepest gratitude goes to my wife, Nadine, and my daughter, Alyssa, for their loving support and understanding during the many hours I couldn’t help out around the house, read books, or play.

Without the help and moral support of these people, this work would not have been possible.

I. INTRODUCTION

A. BACKGROUND

One of the goals of virtual reality is to present to users an interactive environment closely resembling the real world. While not as real as the world we live in, we always strive to make it so, giving the user of the environment some sense that his interactions are real. This matches closely with a goal of military computer simulation: provide as real a training environment as possible so as to maximize the training benefit to the individual and minimize the risks inherent in real world training. With the military's increasing emphasis on joint operations, conducting training exercises in real environments between services, or between different units in the same service, is vitally important, but logistically difficult, if not impossible. This is made more difficult by the ever increasing demand for the U.S. military to conduct extensive operations on unfamiliar foreign soil or waters with little to no advance notice.

NPSNET is a distributed virtual environment and computer simulation research effort which came about in response to these goals and requirements. By being distributed, military units at distant locations can interact with each other in a common and realistic virtual environment, providing useful training between distinct units without the need for being in the same physical location [ZYDA94]. The virtual environment gives the user the feeling of being in the real world with the same freedom that he would have if he was out in the field. This research is an attempt to expand the scope of NPSNET's current implementation, which is primarily geared toward ground-based forces and operations, to include naval helicopter based anti-submarine warfare (ASW) forces and operations.

B. MOTIVATION

1. Expansion of NPSNET’s Scope

Joint operations are becoming the primary focus of the military. For example, land, air, and sea forces from the Army, Navy, Air Force and Marine Corps, and armed services from several foreign countries, were closely integrated in the U.S.-Iraqi War in 1991. Training and rehearsal in a virtual environment representation of the region could be used to prepare the various components of a joint military force by giving them a basic idea of the terrain and what to expect from other services and units during battle. This is not as realistic as actually going into the real world environment of concern, but it is a much more cost effective training tool than assembling all the involved units at the location, whether that be land, sea, or air, or any combination of these.

As already mentioned, NPSNET primarily simulates ground-based military operations. Tanks, foot soldiers, and helicopter gunships represent the bulk of what can currently be simulated. This does not represent the current trend in tactical training and simulation described above. To solve this problem, NPSNET needs to be expanded to involve more platform types and more terrain databases. This would allow a wider variety of military units representing many more operational platform types than currently implemented, to benefit from the joint training that NPSNET can provide. Adding helicopter ASW functionality to NPSNET is a step in this direction.

2. The Need for a Realistic Helicopter Control Model

As stated above, NPSNET is a computer workstation based simulation program of various operational platforms. To provide realistic simulation to users of the system, the user must be able to control the vehicle they are simulating in a manner closely approximating the way they would operate the actual vehicle they use. This maximizes the training benefit of the system by realistically simulating as many aspects of the actual system as practicable, limited by the capabilities of the workstation on which the simulation will run and the input devices utilized.

The current implementation of helicopter operations in NPSNET is based primarily on tactics. An AH-64 helicopter can fire missiles, shoot a machine gun, or drop bombs. With helicopter ASW functionality added to NPSNET, an SH-60 helicopter can drop sonobuoys or torpedoes to find a submarine or fire an anti-ship missile. In both of these cases, however, only half of what is important to a tactical simulation is actually being simulated, which happens to be the tactics themselves. Equally as important to realistic simulation is actual control of the vehicle, in this case the helicopter.

The current model of helicopter control in NPSNET is the same as that for any fixed wing aircraft in the system. While this may provide enough realism and accuracy to the simulation of a fixed wing aircraft, it is not at all realistic for helicopter control, which is more complex. The solution to this problem is to change the control paradigm currently used for helicopters to one that more accurately reflects how a real helicopter is flown. This provides a much greater level of realism to the user, thereby increasing the level of training received.

C. OBJECTIVES

The objective of this research is the design and construction of an interface to simulate helicopter ASW in NPSNET. This interface provides to a user on a single computer workstation or multiple users at different workstations representing one ASW helicopter the capability to realistically prosecute a subsurface contact by utilizing the basic ASW capabilities added to the system. The interface provides the ability to exercise aircrew concepts regardless of whether a single workstation or multiple workstations are participating in the simulation. Another objective is to increase the realism of operating any helicopter in NPSNET by changing the control inputs to inputs representative of a real helicopter.

D. CHAPTER SUMMARY

Chapter II provides an overview of NPSNET, including previous work forming the basis of this research effort. Chapter III describes some basics of helicopter control and crew concept. Chapter IV provides an overview of helicopter based ASW operations. Chapter V details the implementation of helicopter control developed from this research. Chapter VI describes the interface designed for multiple workstation helicopter ASW simulation. Chapter VII provides a description of the use of the designed ASW system. Chapter VIII details results of testing of the system and presents conclusions of this research and recommendations for follow on work.

II. PREVIOUS WORK

A. NPSNET OVERVIEW

The Naval Postgraduate School Networked Vehicle Simulator IV (NPSNET-IV) is a three-dimensional virtual environment and visual simulation system. Using the Distributed Interactive Simulation (DIS) protocol, it operates in real-time over large-scale networks on commercial, off-the-shelf Silicon Graphics workstations [ZYDA95].

In operating NPSNET-IV, a user on a workstation initializes the system to simulate a specific vehicle in a specific terrain database. Currently, the user can choose among a number of aircraft, ground vehicles, or a synthetic human. The choice of terrain database includes a generic training database, a database for Fort Hunter Liggett, California, and a database for Fort Benning, Georgia. Once the system has been initialized, the user has a view of the virtual environment, which is the terrain database selected, from the vehicle chosen. He can then move around in that virtual environment with an amount of freedom based on the properties of the vehicle his workstation is simulating and can fire the weapons that vehicle has been programmed to use.

When a user chooses to initialize the workstation as a helicopter, the functionality provided is that of a ground attack helicopter, either the AH-64 Apache or the NATO designated HIND helicopter. The weapons available include machine guns, missiles and bombs for attacking airborne or ground based entities. The vehicle is moved around in the virtual environment by “flying” the helicopter with the available input devices. These devices include a joystick and throttle, keyboard, or spaceball [ZYDA94]. All of this gives the user the look and feel of being in his helicopter operating inside the virtual environment.

What has been described so far only represents one entity in the virtual environment: one helicopter in a specific terrain database. Another aspect of NPSNET-IV that needs to be described is the network: how other workstations and the vehicles they simulate can interact with the helicopter. Each workstation on the network that participates in the simulation runs its own NPSNET-IV executable. When started, the program searches the local network to see if other workstations are participating in a simulation with the same exercise number. If so, these workstations now participate in the same simulation. For example, if the helicopter was initialized using the same exercise number as another workstation, then these two vehicles can interact with each other. This interaction involves being able to see each other, fire weapons at each other, see explosions caused by each other and their results, or team up with each other against other entities in the simulation.

All of this interaction is possible through the use of the DIS protocol over the network. Each workstation transmits DIS packets to inform other workstations in the simulation about the status of that vehicle. For example, when the helicopter makes a turn, the program sends a DIS packet onto the network indicating a turn. When the other workstations receive this packet, they update their representation of that vehicle so that it executes the same turn as the originator. This same scenario takes place for all dynamic events in NPSNET-IV.

The DIS packets that are sent are called protocol data units (PDU’s). NPSNET-IV uses three of the available PDU types from the DIS protocol. These are the entity state PDU, weapon fire PDU, and detonate PDU [ZYDA94]. All of these indicate to other workstations in the simulation dynamic events that originate on the sending workstation. The result of this communication is that each workstation conducts the same simulation so that interaction among vehicles is as realistic as possible.

Since NPSNET-IV runs over a network, simulating a single vehicle is only part of what it must do. Each workstation must also show its user the other vehicles in the simulation in a realistic manner. To prevent saturation of the network with traffic to constantly update the position of each vehicle in the simulation, each workstation maintains a low resolution model of the other vehicles in the simulation used to calculate the predicted position of those vehicles based on recent PDU’s it received. Each workstation also maintains two models of itself: a high resolution model to track its actual position in the virtual environment, and a low resolution model which uses the same prediction scheme noted above. This method of prediction is called dead-reckoning, and it keeps track of a vehicle’s position, attitude, course and speed by predicting from the data contained in entity state PDU’s [ZYDA94]. When the two vehicle models a workstation maintains of itself differ by a predetermined amount of error, or a certain amount of time has elapsed since the last update, a new entity state PDU is transmitted. This causes the other workstations to update their low resolution model of that vehicle and continue dead-reckoning from the current position stated in the PDU.

As noted earlier, NPSNET-IV is a real-time system. Each workstation bases its simulation on its system clock. So regardless of the graphics frame rate that can be achieved on a particular workstation, the simulation is the same on all workstations. For the dead-reckoning procedures described above then, a workstation receives a PDU to update some other entity in the simulation. Since that PDU is no longer current due to network transmission delays, the workstation dead-reckons from the time listed in the PDU. This time represents the time that the entity state PDU was generated and hence, when the location information about that vehicle was valid. The effect of this dead-reckoning scheme is that one simulation can be conducted on a variety of workstations at once without sacrificing accuracy of the simulation. A lower-end, slower CPU workstation can provide just as much realism in the simulation as one executed on a higher-end, fast CPU and multiple CPU workstation. All workstations show the same vehicles behaving alike and place them in accurate locations since it is all based on time.

The final part of the simulation that must be dealt with is the terrain. In NPSNET-IV, each workstation maintains its own copy of the terrain database. This allows for a high amount of detail when each workstation draws the terrain while at the same time preventing large amounts of network traffic that would be needed if the database were stored in a central location.

B. NPSNET PANEL INTERFACE

In thesis work by Chris McMahan, a windowed subsystem of NPSNET was developed to provide a common interface for the various vehicles of a simulation [MCMA95]. This interface was constructed using Motif and the Viewkit Development Toolkit to separate the functionality of the interface from the simulation and visualization engines. The advantage of this approach is that it allows development of new interface components without sacrificing speed or efficiency in NPSNET. It also allows users to easily adapt to methods of controlling different types of vehicles without having to memorize the many keyboard commands now used to control vehicles.

There are some disadvantages to this approach. First, use of this interface panel detracts from the realism of a simulation. When used as the primary interface for a vehicle in a simulation, a mouse adjusts the various components of the panel, which, in turn, controls the vehicle. Real vehicles use real controls. To simulate these vehicles realistically, similar controls are needed to provide the user with the feel of the vehicle he is simulating. If the panel is used as secondary instrumentation for a vehicle, however, realism can be maintained while still providing more information to the user.

Another disadvantage of the interface employed in McMahan’s research is that there is no concept of one panel controlling one vehicle while another panel controls another vehicle. The reason for this is that the communication between the panel application and NPSNET has no facility for identifying which packet goes to which entity of the simulation. In addition, when NPSNET sends packets to the panel, there is no identifying information about which entity sent the packet. The result of this is that the panel shows information from all entities in the simulation, and any controlling information sent by the panel to the entity goes to any entity that has remote control designated at initialization. Only one panel can effectively be used with one entity, while the rest of the entities have no panel associated with them in the simulation.

Despite these drawbacks, the interface panel concept developed by McMahan is useful to this research as mentioned above: as secondary instrumentation to users of the

ASW helicopter being developed. If the panel can be instantiated once for every ASW helicopter initialized, the door is open for two users to control the helicopter, one using the flight controls and visualization effects already provided by NPSNET, and the other controlling the ASW tactical situation from information provided by the interface panel. If the two users operating these functions can communicate through their respective interfaces, a realistic simulation can be conducted which provides a degree of crew concept needed to effectively prosecute a subsurface contact.

III. BASICS OF HELICOPTER FLIGHT

A. INTRODUCTION

To understand what is needed to make simulation of helicopters more realistic in NPSNET, on must first understand how a real helicopter operates. This chapter describes basic helicopter aerodynamics, helicopter flight control, and the major components of a helicopter. While not intended to be a comprehensive examination of helicopter flight, the information contained here is provided as background for the changes made to NPSNET helicopter simulation, as described in Chapter V.

B. HELICOPTER AERODYNAMICS

There are several techniques for describing the physics of flying helicopters. The most realistic method used today is “blade element theory”, which describes the aerodynamic elements of flight by dealing with the forces on each individual rotor blade [PROU86]. This method for calculating helicopter motion is used in modern flight simulators since it can provide a great deal of realism to the pilot in the simulation. While very complex, these types of simulators accomplish real time calculation and simulation through the use of specialized computing hardware. The design of these computer systems is specifically aimed at fast performance of complex integration to provide the most realism possible to pilots.

The blade element method quickly becomes impractical when considering NPSNET, which must calculate motion for a variety of vehicle types using off-the-shelf computer systems over a networked environment. The comparative slowdowns caused by the use of these systems make the realism provided by the blade element method impossible to recreate. In addition, NPSNET was not designed to provide this level of detail to the individual user.

A simpler method is the “balance of forces” method. This method describes motion resulting from the forces acting on an airframe. The major forces acting on a helicoter airframe are lift, thrust, drag, weight and torque. Representing these forces by vectors gives a much less complex description of aerodynamics in a helicopter, allowing faster computation and a more realistic simulation than is currently accomplished in NPSNET.

Figure 1 shows a general description of the balance of forces on a generic airframe. Assuming in this general case that lift, weight, thrust and drag all act on the aircraft’s center of gravity (C.G.), steady state flight can be maintained if all of these forces cancel each other out. Steady state flight is when there is no acceleration or deceleration, or constant altitude and constant velocity. In the diagram, if the lift vector is equal in length to the weight vector and in the exact opposite direction, the aircraft will not accelerate up or down. The same is true of thrust and drag: if they are equal in length but opposite in direction, the aircraft will not accelerate or decelerate in its forward motion. If both of these cases exist at once, the aircraft will be in steady state flight.

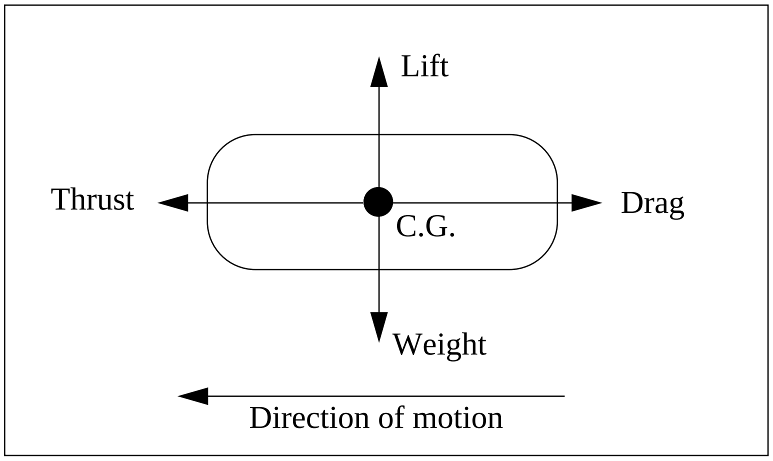


Figure 1: Generic Balance of Forces

In a helicopter, Figure 1 changes slightly. Figure 2 shows how the same forces are represented in a helicopter. The main difference is that lift and thrust emanate from the center of the rotor on top of the aircraft, while weight and drag still originate at the C.G.

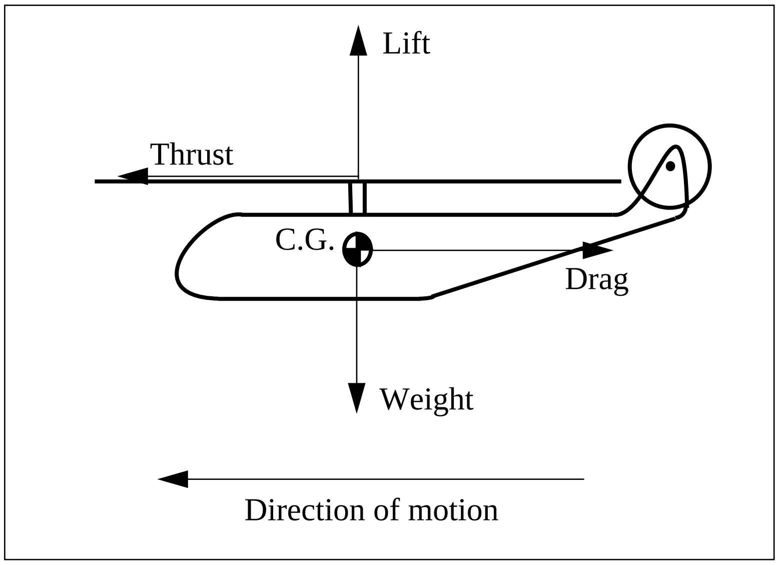


Figure 2: Helicopter Balance of Forces

The following sections give descriptions of each force as they apply to the balance of forces method for describing the aerodynamics of helicopter flight.

1. Weight

The simplest of the forces in Figure 2 is weight. Weight is a force that originates at the C.G. of the helicopter and points to the ground. Weight comes from the following equation:

$$F = m \times a \tag{Eq. 1}$$

where:

F = weight, the weight of the helicopter,

m = mass of the helicopter, and

a = acceleration, the acceleration of gravity.

To overcome the weight of the helicopter and produce flight, the main rotor of the helicopter produces lift. When the amount of lift is greater than the weight, the helicopter will climb. Conversely, when the lift is less than the weight, the helicopter descends. And when the two are equal, altitude is maintained.

In helicopter flight, weight changes as fuel is used during a flight. As more fuel is burned, the helicopter becomes lighter. This results in less power required to maintain steady state flight since the amount of lift needed to offset weight becomes less.

2. Lift

The easiest way to describe lift is by describing how a helicopter hovers. In a helicopter, lift is produced directly by the main rotor. As power is applied by the pilot, more lift is created. When the amount of lift becomes greater than the weight of the helicopter as described above, the helicopter will climb.

The reason this occurs is that each blade of the main rotor is an airfoil, just like the wings of a fixed-wing aircraft. With no power applied, the pitch of each blade is effectively zero, so no lift is produced by the main rotor blades. As the pilot applies power, the pitch of all the blades increases, so lift is produced. If enough power is applied, the rotor blades will produce enough lift to equal the weight, and a hover is produced. For example, if a helicopter weighs 20,000 pounds, the amount of lift required to hover is 20,000 pounds. If less than 20,000 pounds of lift is produced, the helicopter descends. If there is more than 20,000 pounds of lift, it climbs.

Lift is represented by a vector originating from the center of the main rotor. It is perpendicular to the plane of the main rotor blades and points up.

3. Thrust

Helicopters do not produce thrust like a fixed wing aircraft. Jets and propeller airplanes produce thrust by driving a large volume of air rearward while the stationary wings produce lift due to volumes of air flowing over them. In a helicopter, lift is produced when the main rotor blades move through the air fast enough to obtain sufficient airflow over them then drive the airflow downward by pitching the blades up. Thrust, however, must be indirectly produced. The way it is accomplished is by tilting the main rotor from the horizontal plane. This is shown in Figure 3.

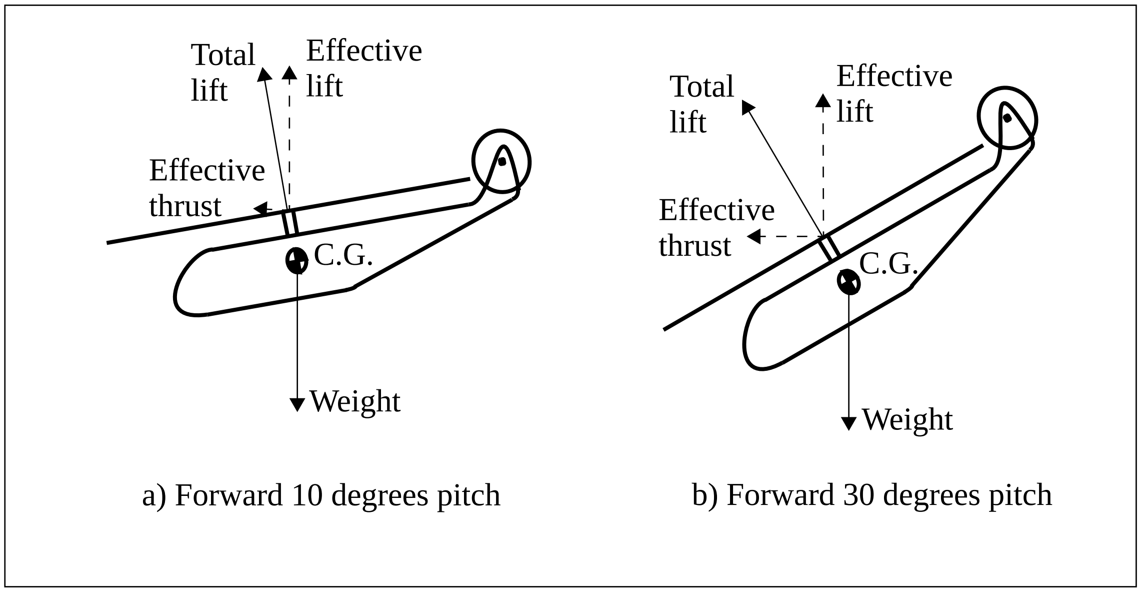


Figure 3: Lift vs. Thrust

As can be seen in Figure 3, the total lift between the two helicopters is the same, as are their weights. The helicopter on the left has less pitch, so the effective lift, the vertical component of total lift, is only slightly less than if there was zero pitch. The horizontal component, effective thrust, has appeared as well. This helicopter will start to accelerate forward from the amount of effective thrust, but will start to descend slightly as the effective lift decreases enough to be unable to counteract weight.

The helicopter on the right in Figure 3 has significantly more forward pitch than the helicopter on the left. The effect here is that there is significantly more effective thrust and less effective lift. The helicopter will accelerate faster and descend more quickly.

One of the unique features of the helicopter is that thrust can be produced in any direction by tilting the main rotor in that direction, including backwards and sideways. This is shown in Figure 4. The importance of this unique quality can be seen in the helicopter’s ability to hover. By tilting the main rotor in the desired direction, the helicopter can be made to fly sideways or even backwards.

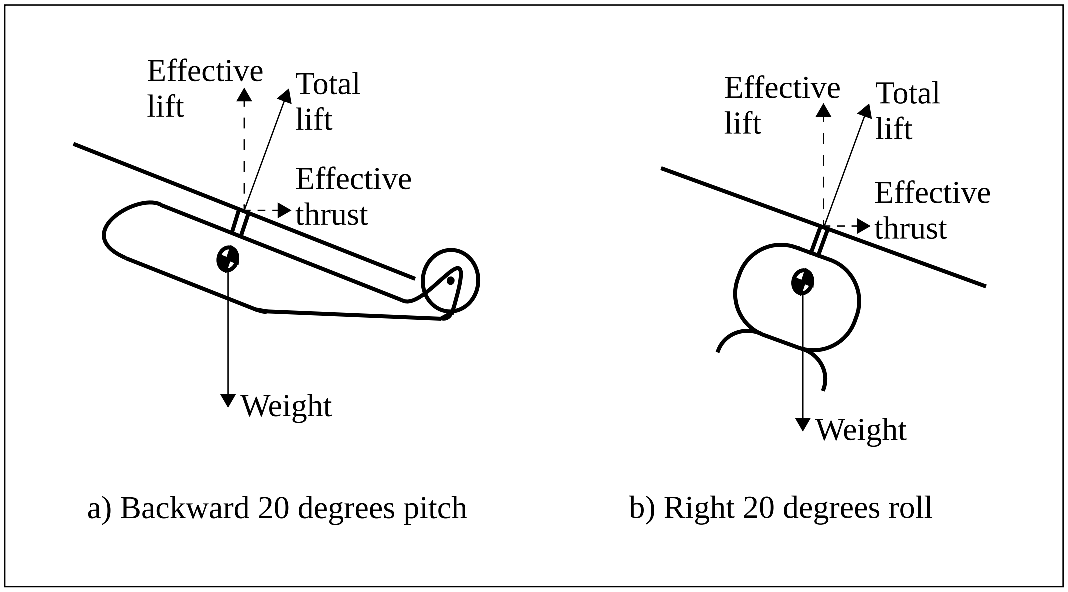


Figure 4: Backwards and Sideways Thrust

4. Drag

As shown in Figure 2, drag is the force that thrust must counteract to provide acceleration and constant state velocity. Drag is opposite to the direction of motion. The amount of drag is dependent on the magnitude and direction of the helicopter's velocity and the angle of attack of the helicopter's fuselage to the oncoming airflow. As velocity increases, drag on the fuselage increases, resulting in the need for more power to go faster. In a similar manner, drag is at a minimum when the fuselage is heading forward. When flying sideways, the cross-sectional area of the fuselage into the airstream is much larger, resulting in higher drag. Therefore, it takes less power to fly forward than it does to fly sideways at the same speed.

5. Torque

Another major force produced by a helicopter is torque, as illustrated in Figure 5. The engines drive a transmission which spins the main rotor in a counterclockwise direction. This spinning causes the fuselage to try to rotate in the opposite direction, clockwise. If this torque force isn't counteracted, the helicopter would spin out of control.

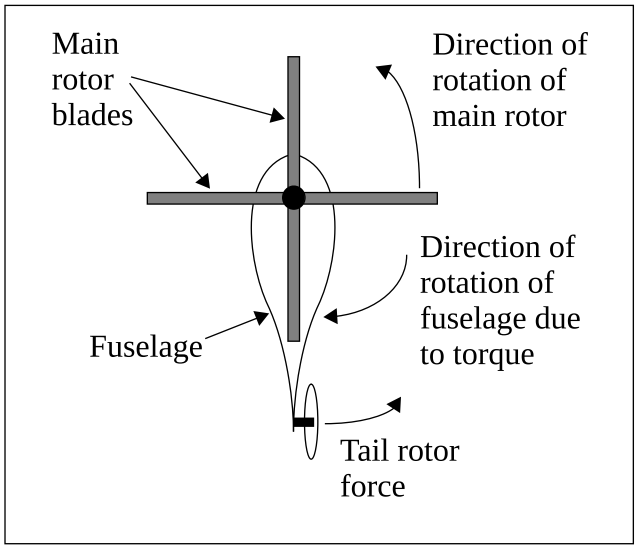

Main rotor blades
Direction of rotation of main rotor
Direction of rotation of fuselage due to torque
Fuselage
Tail rotor force

Figure 5: Helicopter Torque

There are several ways modern helicopters counteract this torque force. The most common way is through the use of a tail rotor. This anti-torque device is mounted on the tail of the helicopter and is oriented parallel, or nearly parallel, to the fuselage, as in Figure 5. The tail rotor produces thrust counterclockwise which pulls the tail end of the fuselage preventing the spinning. So it provides directional control for the fuselage, both in a hover and in forward flight.

Another way that main rotor torque is counteracted is to have two main rotors spinning in opposite directions. Since they spin in opposite directions, the torque that each produces is counteracted by the other, resulting in no torque on the fuselage. A newer method of counteracting torque is the NOTAR system, where a jet engine serves the same function as the tail rotor. Located in approximately the same position as a tail rotor, it produces anti-torque by pushing air away from the direction of rotation desired for the fuselage. This pushes the fuselage in the correct direction to maintain directional control.

C. HELICOPTER CONTROL

Now that all of the aerodynamic forces have been explored, how are these controlled to actually fly a helicopter? This section describes the common controls used by most helicopters: the collective, the cyclic and the rudder pedals. These control devices are shown in Figure 6.

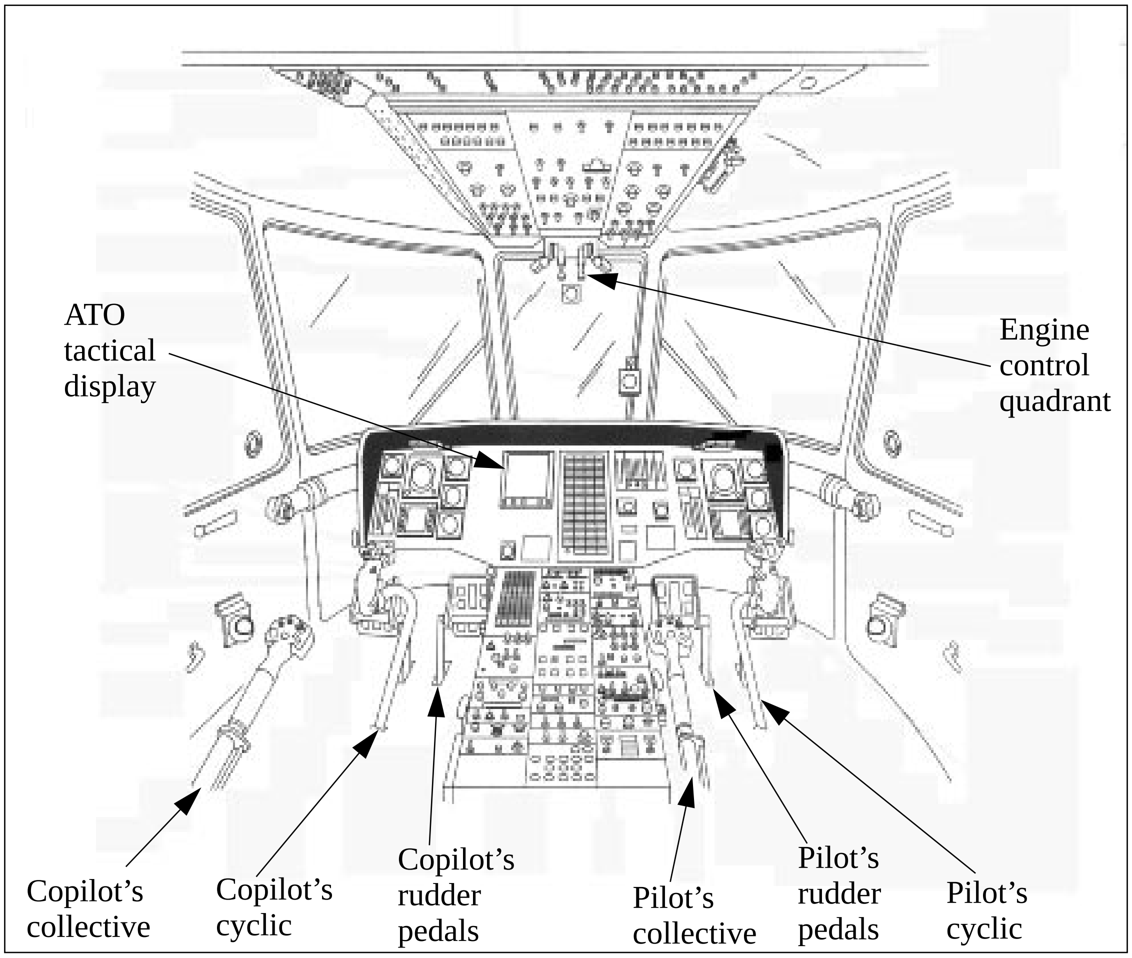


Figure 6: Helicopter Control Components

1. Collective

The collective controls the pitch of the main rotor blades. Controlled by the left hand, the collective is moved up and down to increase or decrease the pitch of the blades all the same amount, or collectively. This increase in pitch increases the lift produced by the main rotor and gives the helicopter the ability to climb. Similarly, moving the collective down decreases the pitch on the main rotor blades producing less or no lift, resulting in a descent.

2. Cyclic

The cyclic provides the pitch and roll control of the helicopter. By moving the cyclic to the left or right, the pitch on each of the main rotor blades is adjusted so that the main rotor disc tilts the same respective direction. Tilting the main rotor in this fashion rolls the fuselage and results in a turn or a slide.

Pitch is controlled in the same way as roll. If the pilot wishes to pitch the nose of the helicopter down, he pushes forward on the cyclic. This causes the main rotor blades to change pitch in cycle, driving the nose of the helicopter down and accelerating the helicopter.

Pitch and roll commands through the cyclic can be combined as well. For example, forward pitch and left roll can be commanded for by moving the cyclic to the forward left position from its center. Any combination of pitch and roll can be input, which tilts the main rotor in the desired direction producing acceleration in that direction.

3. Rudder Pedals

The rudder pedals provide directional control by changing the pitch on the tail rotor, changing the amount of anti-torque as described earlier. Generally, with the rudder pedals in the center position, enough anti-torque is produced for the fuselage to maintain constant heading. Pushing the left pedal forward increases the amount of anti-torque, turning the fuselage to the left. Pushing right pedal turns the helicopter to the right by decreasing anti-torque.

4. Combining Controls

In any aircraft, changing control to produce one desired effect usually requires at least one other control movement to maintain steady flight. One example would be using the collective to start a climb. When the collective is raised, the pitch of the main rotor blades increases to produce more lift. A climb results. One side effect of increasing the collective is a corresponding increase in torque. This is because the pitch of all the main rotor blades increase and more power is required to keep them turning at a constant rate. This results in more torque on the fuselage, so it begins to spin slowly to the right. To counteract this, increases in collective are usually accompanied by adding left pedal. This helps the helicopter maintain a constant heading.

Another example is the relationship between cyclic pitch and collective. If the pilot wishes to accelerate forward but maintain his current altitude, he must push the cyclic forward to tilt the main rotor forward. As described earlier, this increases the effective thrust produced by the main rotor and decreases the effective lift. To maintain the current altitude, the pilot must then increase the amount of effective lift by raising the collective. Finally, the situation described above occurs due to the collective being increased, so left pedal must also be applied to maintain heading.

5. Engine Control

Another concept that should be explained is the role the engines play in helicopter flight. In jet aircraft, the amount of thrust is directly related to the speed of the engines. If the pilot wishes to go faster, he increases the throttle to speed up the engines, which produces more thrust.

In a helicopter, however, more lift is produced by increasing the pitch on the main rotor blades. As was stated earlier, the main rotor blades spin at a generally constant rate. The engines drive a shaft which drives a large reduction gear transmission. This transmission spins the main rotor and the tail rotor. Since it is desired that the main rotor spins at a constant rate, the engines run at a constant speed. Controlling the pitch of the rotors is accomplished through direct linkages to the cyclic, collective, and rudder pedals. The only times that the speed of the engines deviates from their desired constant speed is during start-up, shutdown, and engine malfunction. In normal flight regimes, engine speed is not adjusted, so it is unrelated to control of lift and thrust.

6. Maximum Forward Airspeed in Level Flight

This is another important concept in helicopter flight. There are two limits related to this concept: an aerodynamic limit and a control limit.

The aerodynamic limit is related to the spinning of the main rotor blades and is called blade stall. Blade stall is defined to be the tendency of the retreating blade to stall in forward flight [NTPS90]. Figure 7 shows the difference between the speeds of the main rotor blades relative to the oncoming airflow. The airspeed of a rotor blade is calculated by adding the rotational speed of the rotor blade to the forward airspeed of the helicopter. When the helicopter is in a hover, as in Figure 7a, forward airspeed of the helicopter is zero. Therefore, airspeeds of the advancing and retreating blades are the same. In fact, the airspeed of any blade throughout the 360 degrees of rotation is the same.

In forward flight, however, the advancing blade is rotating into the oncoming airflow while the retreating blade is rotating away from it. Figure 7b shows that the advancing blade has a much faster airspeed than the retreating blade. The airspeed of the retreating blade is calculated by subtracting the forward airspeed of the helicopter from the rotational velocity of the blade since it is opposite in direction to the oncoming airflow. The result is that the retreating blade must increase pitch to maintain the same amount of lift as the advancing blade. If the forward airspeed is too high, the increase in pitch of the retreating blade will cause it to stall, or lose its ability to produce lift. The advancing blade will produce a high amount of lift, even at low blade pitch angles, since its airspeed is so great. Since one side of the aircraft has lift and the other side does not, such an airspeed will cause the helicopter to depart from controlled flight. Due to the fact that the effect of each main rotor blade on the total lift is manifested 90 degrees later, the helicopter nose will pitch up when the retreating blade stalls. This has the effect of decreasing forward airspeed thus increasing the relative airspeed of the retreating blade. Thus, the stall condition is automatically corrected. However, the forces on the main rotor system during this transition from normal flight to stall and back can result in catastrophic damage. Hence, helicopter flight manuals limit the forward airspeeds that a pilot is allowed to fly so that this blade stall condition is avoided. [NTPS90]

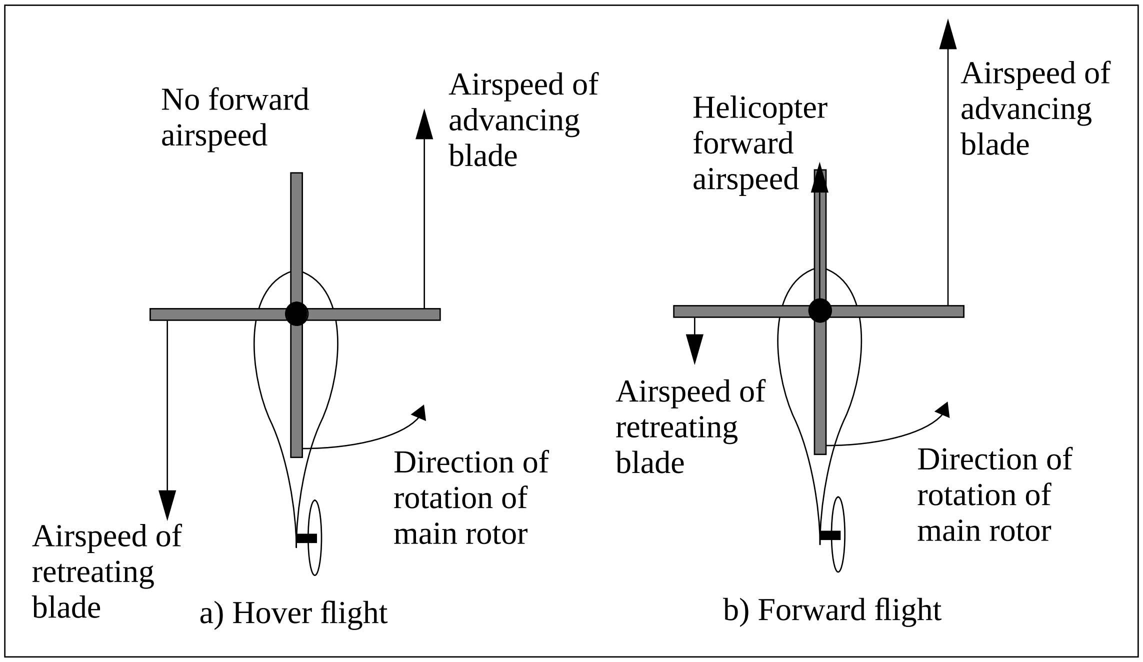

No forward airspeed
Airspeed of advancing blade
Airspeed of retreating blade
Direction of rotation of main rotor
a) Hover flight
Helicopter forward airspeed
Airspeed of advancing blade
Airspeed of retreating blade
Direction of rotation of main rotor
b) Forward flight

Figure 7: Main Rotor Blade Airspeed

The other limitation on forward airspeed is a result of control limits. As forward airspeed increases, the pilot must increase collective to maintain altitude, as described earlier. If the helicopter is heavy enough, the collective will reach the upper limit of its control range before blade stall airspeed is reached. In this case, maximum forward airspeed in level flight is limited by the physical control movement limit instead of the aerodynamic limit of blade stall.

D. HELICOPTER COMPONENTS

All modern powered aircraft are made up of components necessary to stay airborne in level flight. These consist of the fuselage, engines, some type of airfoils, and the cockpit. These components are present in fixed-wing aircraft and helicopters, but their structure and function can differ. Figure 8 shows these components on a helicopter.

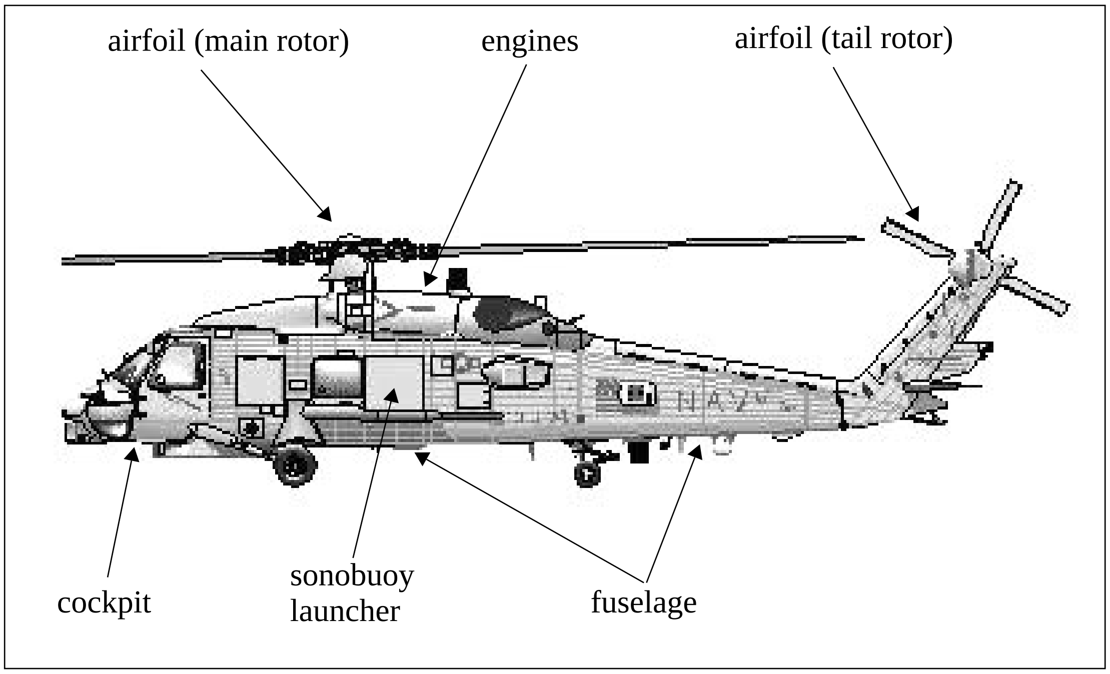


Figure 8: Helicopter Components

The purpose of a fuselage is the same for a helicopter and fixed-wing airplane: to house crewmembers and cargo, and to serve as an attachment point for the external components of the aircraft. In a helicopter, the engines, rotors, and transmissions are all attached to the fuselage. In addition, all electrical components and control linkages are inside the fuselage.

The next major component of a helicopter is the engine. This is usually a gas-turbine, or "jet" engine, and many helicopters have two or more engines. The purpose of the engines is to drive the thrust producing component of aircraft. In a jet airplane, the thrust producer are turbines internal to the engine, and in a propeller airplane, it is the propeller. In a helicopter, this is the rotor system.

The rotor system is a component that is unique to helicopters. Most helicopters have a rotor system composed of a main rotor and a tail rotor. The main rotor is a series of large blades that spin to produce both lift and thrust. When the pitch on the blades is increased, the blades produce more lift. When the blades are tilted, thrust is produced in the direction of the tilt.

The tail rotor is the anti-torque device on a helicopter. By pulling/pushing the tail of the fuselage opposite the direction resulting from the torque produced by the main rotor blades, directional control of the helicopter is achieved.

Finally, the cockpit is the location where the pilots sit to control the aircraft. An example of a cockpit is shown in Figure 6. The cyclic, collective, and rudder pedals are located in the cockpit, with duplicate sets for the pilot and copilot. Both the pilot and copilot have full control capability over the helicopter. In addition to the flight controls, other components located in the cockpit are flight and engine instruments, electrical controls and circuit breakers, and tactical controls. The cockpit contains ample windows for the pilots to see forward, up, down, and to the side. This allows them to fly in any direction and land in any suitable location.

E. SUMMARY

Although many of the key concepts of helicopter aerodynamics and flight have been introduced here, much of the detail has been left out. Only basic concepts were addressed in the context of NPSNET, so only those concepts were explained in any detail. More complex concepts, like extreme aerodynamic situations, emergency procedures, and autorotations, were not introduced as they are beyond the scope of this research.

IV. BASICS OF HELICOPTER ASW

A. INTRODUCTION

The other portion of this research involves simulating a helicopter conducting an anti-submarine warfare (ASW) mission. To understand the simulation system that is implemented, this chapter describes the components of an ASW mission: the flightcrew, the sensors, and the weapons. The descriptions given are specific to the SH-60B helicopter, but generally apply to most modern ASW helicopters.

B. HELICOPTER ASW FLIGHTCREW

The flightcrew of an ASW mission in an SH-60B helicopter consists of three members: the pilot, the Airborne Tactical Officer, and the Sensor Operator. Each has specific responsibilities in an ASW mission.

1. Pilot

The pilot performs the flight maneuvers of a mission. His primary responsibility is the safe conduct of the mission, and he must coordinate the tactics used during the flight with the ATO. [LWSM92]

2. Airborne Tactical Officer (ATO)

The primary responsibility of the ATO is the tactical conduct of the ASW mission. This includes deciding where to deploy sonobuoys, maintaining tracks on submarine contacts, managing use of all helicopter sensors, navigation to target areas for employment of sensors and weapons, directing weapon delivery and evaluation of attacks. Another important responsibility is to back up the pilot in safe conduct of the mission. [LWSM92]

3. Sensor Operator (SO)

The SO's primary responsibility is to operate the aircraft's sensors and monitor the status of the system. He controls the radar/IFF system, analyzes data from any sonobuoys, controls the MAD system and analyzes its data. [LWSM92]

4. Aircrew Coordination

One of the most important aspects of safe and successful completion of an ASW mission is aircrew coordination. The concept is based on each member of the flightcrew taking care of his own responsibilities during the mission while supporting the other members of the flightcrew in their responsibilities.

Crew coordination works like this: the pilot operates the flight controls as his primary duty. He also helps the ATO in his duties by keeping the helicopter at an optimum altitude and location to accomplish the next task of the mission. This is so sonobuoys or torpedoes can be launched quickly when needed. The pilot also maintains a lookout for possible visual contact with a submarine, either by seeing one underwater or by seeing its periscope.

The ATO controls the conduct of the tactical portion of the mission. For his part of crew coordination, he backs up the pilot on the flight instruments so that safety of flight is maintained. If the pilot flies the helicopter into an unsafe situation, the ATO is responsible for notifying the pilot or taking the control of the helicopter to get it back in a safe flight regime.

In addition to operating the helicopter's sensors, the SO aids the ATO by giving advice about the deployment of sonobuoys and weapons based on the information being received by the sensors. He also informs the pilot and ATO of equipment malfunctions or failures that affect the mission or the safety of flight.

C. HELICOPTER ASW SENSORS

The SH-60B helicopter sensors for use in ASW missions are sonobuoys, the radar/ Identification Friend or Foe system, and the magnetic anomaly detector system. This section gives a brief description of each of these.

1. Sonobuoys

The primary ASW sensor for the SH-60B is the sonobuoy. A sonobuoy is an acoustic underwater sensor used for detecting submarines. The sonobuoy is a cylinder about forty inches long and eight inches in diameter. When launched from the helicopter in-flight, it falls until it hits the water, at which point the cylinder deploys a float to stay on the surface of the water. This float contains a radio transmitter for transmitting information back to the helicopter. The sonobuoy also deploys an underwater microphone, called a hydrophone, several hundred feet below it to listen for acoustic signals.

There are two kinds of sonobuoys used by the SH-60B helicopter: active sonobuoys and passive sonobuoys. The passive sonobuoy can only receive acoustic signals, as shown in Figure 9. Any acoustic data received by the hydrophone is transmitted back to the helicopter via the radio transmitter. Computers in the helicopter analyze the signals and transform them into usable information for the aircrew. The information tells the aircrew the bearing from the sonobuoy to the subsurface contact. [LWSM92]

An active sonobuoy transmits a sonar signal from its hydrophone into the water, then listens for the echo of that signal. The echo of the sonar signal is received when it is reflected off of an underwater contact back to the hydrophone as in Figure 10. The data is transmitted back to the helicopter via the radio transmitter and, when analyzed, gives range and bearing information from the sonobuoy to the submarine. [LWSM92]

The SH-60B helicopter can carry and launch any combination of up to 25 active or passive sonobuoys. The sonobuoy launcher is located on the left side of the fuselage, as seen in Figure 8. [LWSM92]

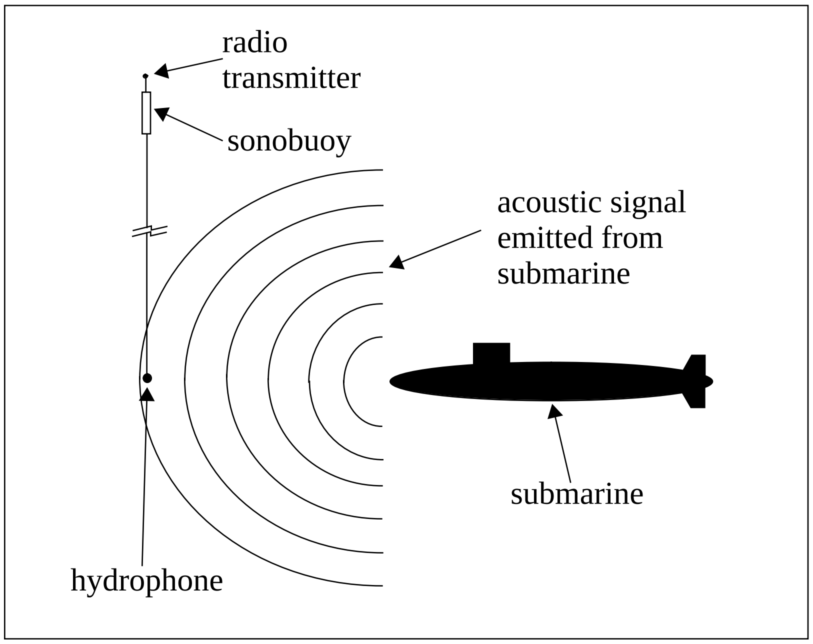


Figure 9: Passive Sonobuoy Operation

2. Radar and Identification Friend or Foe (IFF) System

The radar system is used to detect surface or airborne contacts [LWSM92]. In the ASW mission, the radar is used to detect surfaced submarines or enemy ships.

The IFF system is located on the radar and is used to identify whether potential threats are from enemy forces or friendly forces [LWSM92]. The system, like the radar, does not have underwater capability, so its use in ASW is limited to identification of airborne and surfaced contacts, including surfaced submarines.

3. Magnetic Anomaly Detector (MAD)

The MAD system is designed to detect submarines by measuring the earth's magnetic field. It accomplishes this by detecting disturbances in the earth's magnetic field caused by a submarine. The system then gives the aircrew a location at which the disturbance occurred. [LWSM92]

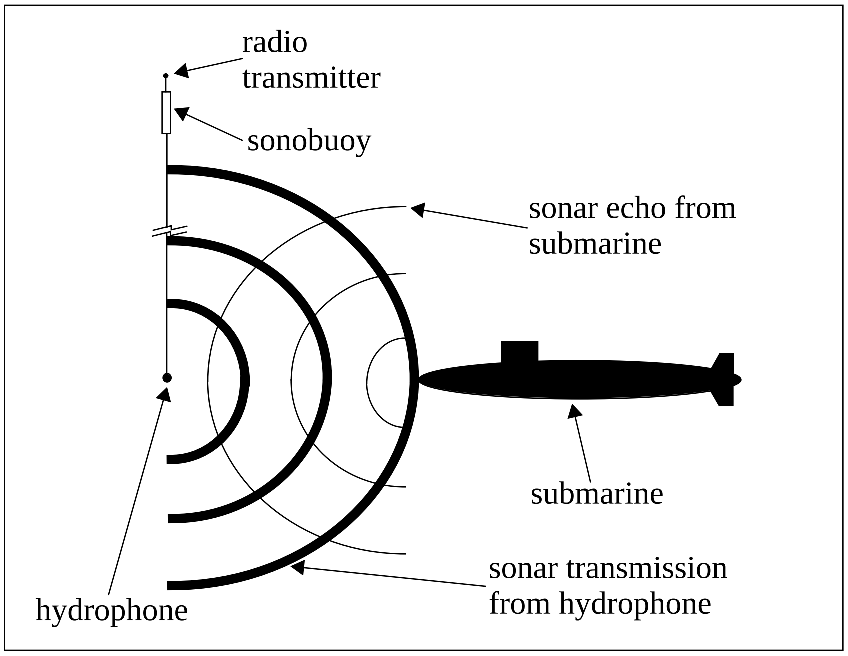


Figure 10: Active Sonobuoy Operation

D. HELICOPTER ASW WEAPONS

1. Mk 46 and Mk 50 Torpedoes

The torpedo is the primary ASW weapon used by the SH-60B helicopter. The torpedo is launched from pylons on either side of the helicopter fuselage. When launched, the torpedo falls unguided to the water. After water impact, it uses active sonar to home in on subsurface contacts. The launch point of the torpedo is determined by the ATO based on his track of the submarine and the capabilities of the torpedo that will be launched. [LWSM92]

Details about search patterns, acquisition ranges, run times and speeds of either the Mk 46 or Mk 50 torpedo are classified and beyond the scope of this research. The description given above is general and the extent of what will be simulated as a result of this research.

2. Penguin Missile

The only other ASW weapon capable of being carried on the SH-60B is the Penguin missile. This is an infrared homing, air-to-surface missile. Launched from pylons on the fuselage, the missile is a “fire and forget” weapon, meaning that once it is fired, it is not dependent on the helicopter for any information or guidance. Prior to launch, though, the missile receives aircraft velocity, heading and altitude information and targeting information from the SH-60B weapon systems. This information is used by the missile to align its inertial navigation system and to navigate to the target unaided. [LWSM92]

E. SUMMARY

This chapter describes the tools used in prosecuting an ASW threat. Most of the actual tactics involved are classified and beyond the scope of this research. The purpose here was to give a basic knowledge of the personnel, sensors, and weapons needed to successfully track and attack a subsurface contact so that they could be reasonable simulated in NPSNET. Implementation of classified tactics and sensor/weapons is independent of the system developed for NPSNET by this research.

V. HELICOPTER CONTROL MODELLING IN NPSNET

A. PREVIOUS HELICOPTER CONTROL METHODOLOGY

Motion of helicopters in NPSNET prior to this research was based on a single vector representing the velocity of the helicopter. The vector is oriented from the center of the helicopter through its front, as shown in Figure 11. Its direction in NPSNET world coordinates is calculated from the heading, pitch, and roll angle values as determined from joystick input. Always being oriented out of the front of the helicopter, however, means that the it is limited to motion forwards and backwards.

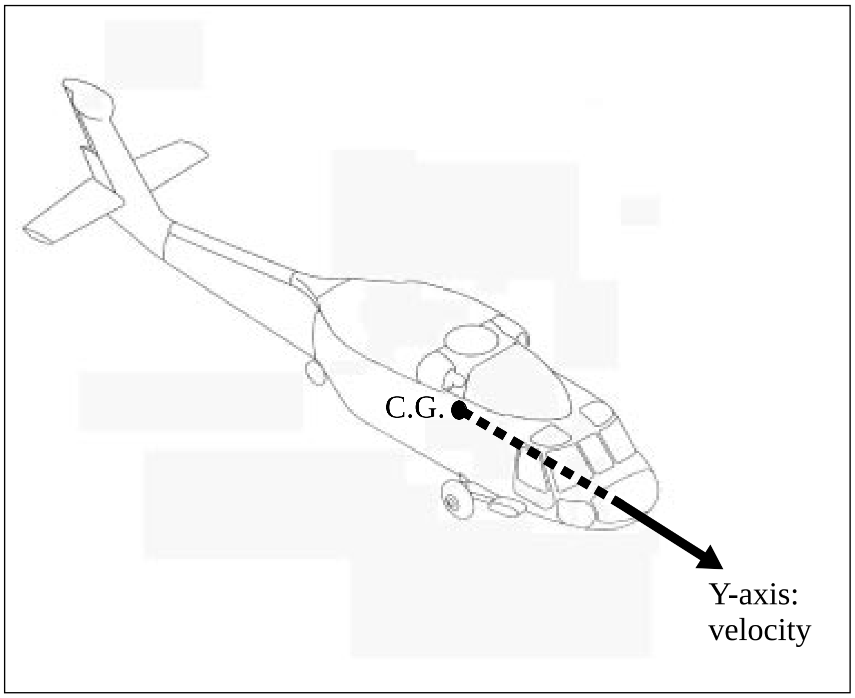


Figure 11: One Vector Control Model

The speed of the helicopter, represented by the velocity vector’s magnitude, is determined by the throttle setting. It can vary anywhere from zero to the maximum speed of the helicopter, defined at compile time to be 100 m.p.h., regardless of its attitude. This further limits the helicopter to forward motion without the capability of going backwards, something real helicopters are quite capable of.

Another problem here is that this model allows the simulated helicopter to remain stationary in space, simulating a hover, regardless of its attitude. As explained in Chapter III, however, real helicopters cannot hover at pitch and roll settings other than zero. A helicopter must first decelerate to zero airspeed then set its hover attitude. Any other pitch and roll setting causes the helicopter to accelerate out of a hover.

The result of this model is that the helicopter has freedom of motion in pitch, roll, and heading angles, and in its positive y axis only. A new model was needed to accurately model how real helicopters can move.

B. NEW HELICOPTER CONTROL METHODOLOGY

This section describes the methodology for calculating the helicopter’s motion. It describes how the control data from the joystick, throttle and rudder pedals is interpreted to calculate velocity.

Real helicopter motion, described in Chapter III, has freedom of motion in pitch, roll, and heading angles as well as positive and negative x, y, and z axes. This allows the helicopter to slide left and right (x axis), go forward and backward (y axis), and go up and down (z axis). This unique quality of helicopters allows independent motion in any of these axes.

To accurately model this ability, a four axis system was developed. This model, shown in Figure 11, uses control inputs to calculate the speeds in the x, y, and z axes, and is based on the balance of forces theory of aerodynamics described in Chapter III. The first step is to extract the control input data and convert that into magnitudes for the four vectors and into heading pitch and roll angles.

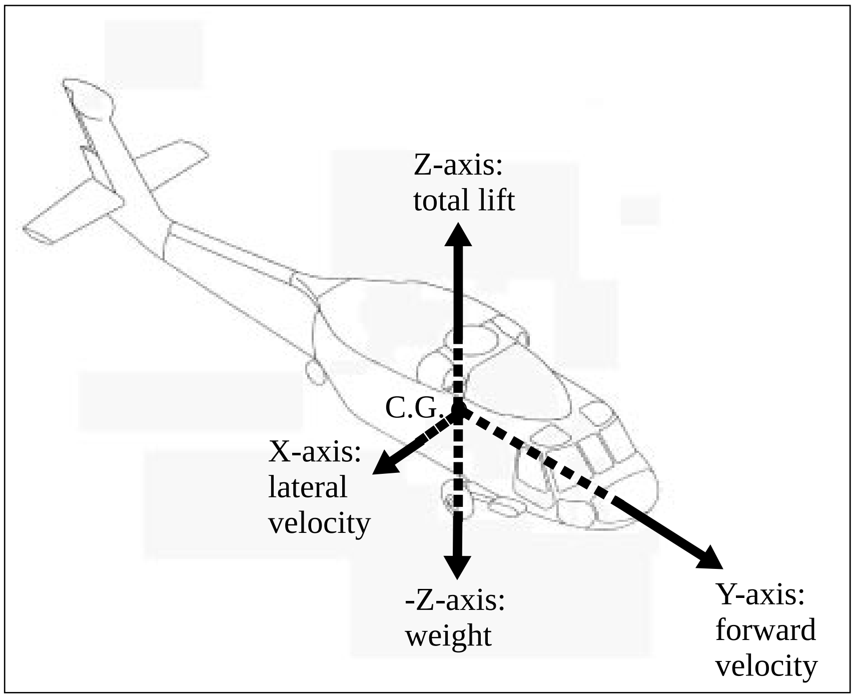


Figure 12: Four Vector Control Model

1. Flight Control Data

a. Joystick Input

Figure 13 shows how the raw data from the joystick gets converted into values for the x and y axes in Figure 11. The joystick is used to simulate a helicopter cyclic, which controls the helicopter's pitch and roll. Raw data from the joystick ranges from -1.0 to 1.0 in both the pitch and roll channels. In the pitch channel, -1.0 is returned by the joystick when it is pushed all the way forward, while 1.0 is returned when it is pulled back. The center position is 0.0, and values in this range indicate how far forward or backward the joystick is moved. The roll channel operates in a similar manner: -1.0 is the leftmost position, 0.0 is the center, and 1.0 is the rightmost position, with varying values in between.

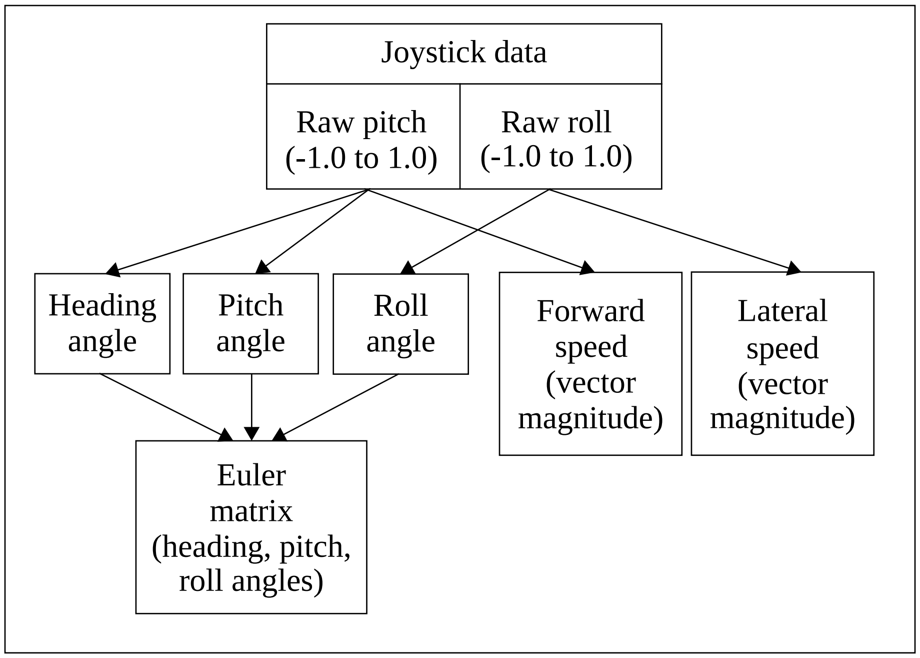

Joystick data
Raw pitch (-1.0 to 1.0)
Raw roll (-1.0 to 1.0)
Heading angle
Pitch angle
Roll angle
Forward speed (vector magnitude)
Lateral speed (vector magnitude)
Euler matrix (heading, pitch, roll angles)

Figure 13: Joystick Data Flow

The raw pitch and roll data is used to find the magnitude and then the direction of the forward and lateral velocity vectors. In calculating the vector magnitudes, raw pitch and roll data are first used to calculate forward and lateral change variables. The equations used are:

$$\Delta\text{forward} = -(\text{raw pitch data} * 0.09) \qquad \text{(Eq. 2)}$$

$$\Delta\text{lateral} = \text{raw roll data} * 0.09 \qquad \text{(Eq. 3)}$$

These equations represent the amount of change that will be applied to acceleration in each of these channels. For example, assume that the helicopter is at a constant speed with the joystick in the center position (raw pitch and roll are both 0.0). If the pilot wishes to decelerate, he pulls back on the joystick. The raw pitch data is now some value between 0.0 and 1.0, depending on how far the joystick was pulled back. As can be seen in Eq. 2, this value is multiplied by the dampening constant 0.09 then negated, which results in a value between -1.0 and 0.0. This change variable is later added to forward acceleration, indicating that the acceleration should get smaller. In the case of the example, since the helicopter was at constant airspeed, there was no acceleration, so acceleration becomes negative and the helicopter decelerates as a result. This works in a similar manner for lateral acceleration, with the exception that the value is not negated, as shown in Eq. 3. The reason for this is that when the joystick is placed to the right of its center position, the raw roll data ranges between 0.0 and 1.0. This would indicate that the pilot wishes to accelerate to the right. The helicopter's x axis is also positive on the right. So, the lateral change data is directly related to lateral acceleration.

The next step in calculating the magnitude of the forward and lateral velocity vectors is to input the change values calculated above into the acceleration equations. These equations are as follows:

$$\text{forward accel} = \text{forward accel} + (\Delta\ \text{forward} * \cos(\text{roll angle})) \qquad \text{(Eq. 4)}$$

$$\text{lateral accel} = \text{lateral accel} + \Delta\ \text{lateral} \qquad \text{(Eq. 5)}$$

$$\text{lateral accel} = \text{lateral accel} + (\Delta\ \text{lateral} / \text{forward speed}) \qquad \text{(Eq. 6)}$$

In the example above, the forward change variable is negative since the pilot wishes to decelerate. As can be seen in Eq. 4, this value is multiplied by the cosine of the roll angle. This angle was calculated in the previous display frame and is used as a starting point for calculating the roll angle in the current frame. The cosine of the roll angle is multiplied by the forward change amount to reduce the amount of acceleration/deceleration as angle of bank increases. Angle of bank is a way to describe roll angles without the use of negative values. For example, 30 degrees of roll is the same as 30 degrees right angle of bank. Similarly, -30 degrees of roll is 30 degrees left angle of bank.

To show why this cosine calculation is needed in the forward acceleration calculation, assume the helicopter is rolled to the right 90 degrees. To increase the rate of turn, the pilot pulls back on the joystick. This is a common maneuver in forward flight: roll into an angle of bank, then pull back on the cyclic to increase the rate of turn. If the cosine calculation isn't made, forward acceleration will decrease and the helicopter will decelerate rapidly. In a real helicopter, this does not occur due to the fact that, at 90 degrees angle of bank, pulling back on the joystick will not result in the helicopter trying to climb. It only increases how fast it turns. By taking the cosine of the roll angle and multiplying it by the forward change previously calculated, the effect of the forward change variable is decreased as roll angle increases.

Once the forward change value is multiplied by the cosine of the roll angle, this value is added to the forward acceleration value from the previous display frame. The effect here is that acceleration of the helicopter is smooth and reacts slowly to the input changes by the pilot.

Calculation of lateral acceleration operates in a similar manner to forward acceleration except that there is no adjustment made as a result of roll. In Eq. 5, the lateral change amount has its full effect on acceleration when the helicopter is flying slower than 25 miles per hour (m.p.h.). So, at low airspeeds, the helicopter can accelerate into a left or right slide at its maximum rate. As airspeed increases above 25 m.p.h., however, the effect of the lateral change amount on lateral acceleration is decreased by the amount of forward airspeed, as shown in Eq. 6. This simulates the airstream acting on the fuselage at higher forward speeds and thereby restricting lateral acceleration. Above forward speeds of 50 m.p.h., lateral speed and acceleration are set to zero to make the helicopter easier to fly for non-helicopter pilots.

To complete the calculation of the forward and lateral vector magnitudes, the forward and lateral accelerations, respectively, are dampened and added to the magnitudes from the previous display frame. The equations for calculating the magnitudes, which is the speed the helicopter will travel in its x and y axes, are as follows:

$$\text{forward speed} = \text{forward speed} + (\text{forward accel} * 0.1) \qquad \text{(Eq. 7)}$$

$$\text{lateral speed} = \text{lateral speed} + (\text{lateral accel} * 0.1) \qquad \text{(Eq. 8)}$$

Acceleration values are dampened in Eq. 7 and Eq. 8 before being added to previously calculated speed values to provide realistic helicopter acceleration based on control inputs. The magnitudes achieved from these equations are later used to scale the forward and lateral vectors, respectively, for ultimately calculating the helicopter's velocity.

As seen in Figure 13, the joystick data is also used to calculate heading, pitch and roll angles based on their values from previous display frames. No significant changes were made to these calculations from the previous helicopter control model as the angles calculated appear realistic enough for the model resulting from this research. The only change to these angle calculations was to add the capability to use rudder pedals, as explained in the following section.

b. Rudder Pedal Input

Raw data from the rudder control pedals varies from -1.0 (left pedal full forward) to 0.0 (pedals centered) to 1.0 (right pedal full forward). This data is used to calculate heading only, as shown in Figure 14. On the ground or at slow airspeeds (less than 25 m.p.h.), raw data from the rudder pedals manipulates the heading angle from the previous display frame. This allows the helicopter to rotate around its z axis.

At airspeeds higher than 25 m.p.h., raw rudder information is combined with heading and roll angle data, raw pitch data from the joystick, and the calculated forward airspeed to calculate the new heading. Most elements of this calculation are from the original helicopter control model. The last part of the equation (raw rudder data/ airspeed) has the effect of changing the heading based on how far a rudder pedal is pushed in. Pushing the left pedal forward halfway, for example, causes the helicopter to execute a moderate rate of turn to the left. Pushing the left pedal all the way forward, however.

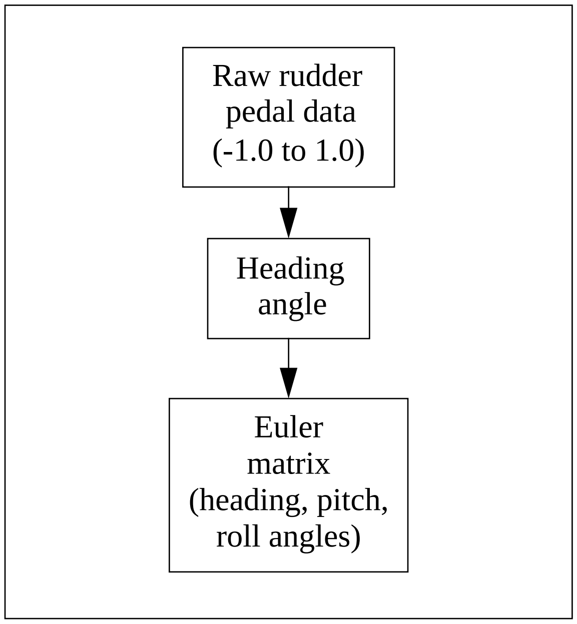

Raw rudder pedal data (-1.0 to 1.0)
Heading angle
Euler matrix (heading, pitch, roll angles)

Figure 14: Rudder Pedal Data Flow

Dividing the raw rudder data by the calculated forward speed reduces the rate of turn as forward airspeed increases. For example, if the left pedal is pushed all the way forward at 30 m.p.h., the helicopter executes a moderate turn to the left. Then the helicopter is flying at 180 m.p.h. though, the same pedal position results in a very slow rate of turn.

c. Throttle Input

Raw throttle data is used to calculate the amount of thrust produced by the main rotor of the helicopter, which represents the magnitude of the thrust vector, as shown in Figure 15. The data ranges from 0.0 in the rearward position to 1.0 in the forward position. This data is multiplied by a constant representing the maximum amount of thrust can be achieved, as shown in the following equation:

$$\text{hrust setting} = (\text{raw throttle} * 40{,}000) - (\text{forward speed} * 111.1) \qquad \text{(Eq. 9)}$$

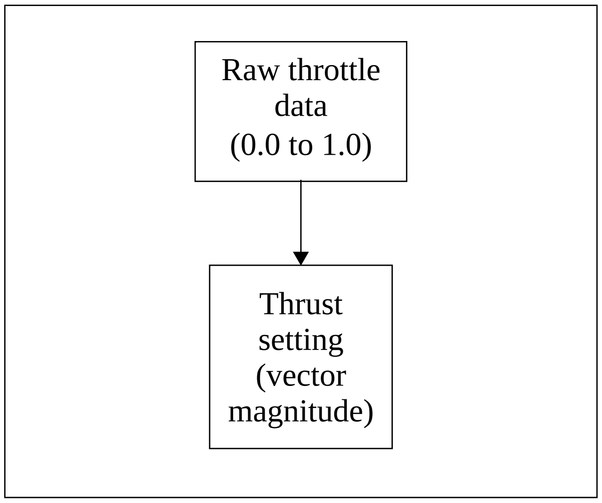

Raw throttle data (0.0 to 1.0)
Thrust setting (vector magnitude)

Figure 15: Throttle Data Flow

By multiplying the raw throttle data by 40,000, the maximum thrust available, the thrust produced is then a percentage of the maximum. For example, having the throttle at the halfway position results in raw throttle data of 0.5. After multiplying this by the maximum thrust available, the thrust produced is 20,000 pounds.

The additional adjustment made to the thrust setting, shown in Eq. 9, reduces the amount of thrust produced as airspeed increases. This is done to simulate maximum forward airspeed in level flight. The SH-60B can achieve about 180 m.p.h. in normal level flight. To simulate maximum forward airspeed in level flight, the thrust produced by the helicopter must equal the weight at the maximum airspeed. This condition will result in the helicopter being flown at maximum speed with maximum power applied while maintaining its current altitude. If the pilot flies the helicopter any faster by pushing the joystick forward to pitch the nose down, the resultant thrust produced will decrease to less than 20,000 pounds. Therefore, the helicopter will descend. Also, since the throttle is already at maximum, a climb cannot be initiated by adding power to produce more thrust. The only way to climb in this situation is to pull back on the joystick thereby pitching the helicopter's nose up. The helicopter will climb from its increase in angle of attack to the oncoming airflow, but airspeed will also decrease.

To reduce the thrust produced at maximum forward airspeed to a point where it equals the weight of the helicopter, a constant factor, when multiplied by the airspeed, is subtracted from the thrust calculation resulting from the throttle position. This is shown in Eq. 9. The constant 111.1 produces this result. It is calculated from the following equation:

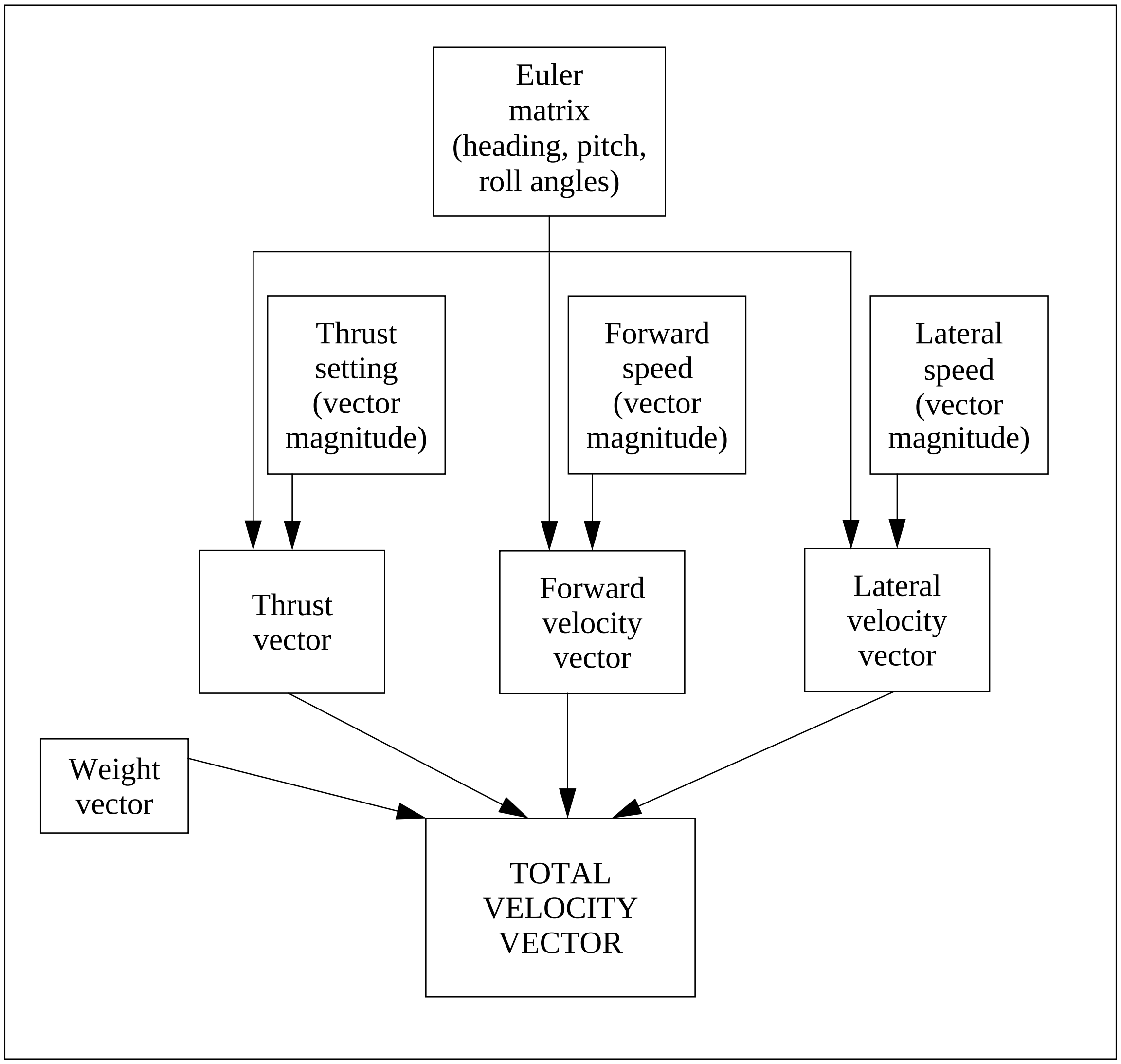
$$\text{constant} = \frac{\text{helicopter weight (lbs)}}{\text{maximum forward speed (mph)}} \qquad \text{(Eq. 10)}$$

This constant affects flight at any airspeed, not just flight at maximum forward level flight airspeed. The effect this calculation has on helicopter control is that as it flies at a faster airspeed, more power is required to maintain altitude. This adds realism to the simulation of the helicopter as well as limit the forward speed in level flight.

2. Vector Calculations

Once the magnitudes of the forward speed, lateral speed and thrust vectors in Figure 11 have been calculated based on control inputs, their directions need to be calculated. This is accomplished using the heading, pitch and roll angle calculations and the Performer function *pfMakeEulerMat()*. This function takes the heading, pitch and roll angles and transforms them into a 4-by-4 matrix representing the amount of rotation in world coordinates about an object's x, y and z axes [HART94].

Once the matrix is formed, another Performer function, *pfXformPt3()*, is used to transform the matrix into the forward velocity, lateral velocity, and total lift vectors, shown in Figure 16. This function transforms a vector expressed in the object's coordinate system into a vector expressed in world coordinates [HART94]. It is used here to transform the x, y and z axes in the helicopter's coordinate system to the same axes expressed in world coordinates. The helicopter's x axis is the lateral velocity vector, its y axis is the forward velocity vector, and its z axis is the thrust vector.



Euler matrix (heading, pitch, roll angles)
Thrust setting (vector magnitude)
Forward speed (vector magnitude)
Lateral speed (vector magnitude)
Thrust vector
Forward velocity vector
Lateral velocity vector
Weight vector
TOTAL VELOCITY VECTOR

Figure 16: Total Velocity Vector

The next step is to normalize these vectors using the Performer function *pfNormalizeVec3()*. This function scales a vector to unit length [HART94]. It is used here to make the forward velocity, lateral velocity, and thrust vectors a uniform length to then scale them to the desired magnitudes as previously calculated.

Next, each vector is scaled to its respective magnitude, shown in Figure 16, using the *pfScaleVec3()* Performer function. The thrust vector is scaled to the thrust setting, the forward velocity vector is scaled to the forward speed, and the lateral velocity vector is scaled to lateral speed. These vectors now represent their respective vectors in Figure 11.

3. Weight Vector

The fourth vector in Figure 11 is the weight vector. This vector always points down from the center of the helicopter in the negative z axis of the world coordinate system. This is accomplished by using *pfScaleVec3()* to scale the world's z axis vector of unit length, which is the vector <0, 0, -1>, to the constant representing the helicopter's weight, which is set to 20,000 pounds. Since this vector always points to the ground, it is not transformed from the 4-by-4 Euler matrix.

4. Velocity

The final step in calculating the helicopter's motion is to calculate the velocity vector. This is done by adding the thrust, weight, forward velocity and lateral velocity vectors to obtain the total speed and direction of the helicopter.

The first step here is to use the Performer function *pfAddVec3()* to add the forward velocity and lateral velocity vectors into a temporary vector. This represents the velocity in the helicopter's x-y plane and is later added to the thrust and weight vectors to obtain total velocity

This same function is used to add the thrust and weight vectors. The result represents the total force produced by the main rotor. This resultant vector, labeled total lift, determines whether the helicopter climbs or descends. Since the helicopter's pitch and roll angles determine the direction of the total lift vector, as shown in Figure 3, its vertical component (effective lift), must counteract the helicopter's weight. If the effective lift is greater than the helicopter's weight, it will climb; if it is less than the weight, the helicopter descends. If they are the same, the helicopter will maintain its altitude.

The next step is to scale the total lift vector just calculated using *pfScaleVec3()*. The total lift vector represents the force produced by the helicopter's main rotor. To translate force into acceleration, Eq. 1 is used. To translate the total lift vector into acceleration, it must be scaled by the inverse of the helicopter's mass, a constant defined at compile time. This has the effect of shortening the total lift vector without changing its direction. This resultant vector is scaled again with *pfScaleVec3()* by Δ time, the elapsed time between the last display frame and the current one, to change acceleration into velocity. The total lift vector now represents the motion of the helicopter resulting from the lift produced by the helicopter to counteract its weight.

Finally, the temporary vector representing the helicopter's x-y plane velocity and the total lift vectors are added using *pfAddVec3()*. This velocity is then used to calculate the helicopter's position in the world coordinate system for the next display frame.

To make the helicopter easier to fly, limits were programmed into this system to prevent the helicopter from flying to fast sideways or backwards. An SH-60B is limited to flying at about 40 m.p.h. in sideways or rearward flight [NTPS90]. The reason for this limit is to prevent damage to the helicopter resulting from extreme aerodynamic situations. The programmed limit is therefore 40 m.p.h. for both sideways and rearward flight. Limiting the program makes the simulated helicopter easier to control for inexperienced users and prevents flight in unusual aerodynamic situations, which are beyond the scope of this research.

5. Ground Motion

The main difference between calculation of motion when the helicopter is on the ground versus when it is in the air is that there can be no sideways motion. This is due to the helicopter's landing gear, which only allows it to travel forwards and backwards. As a result, no lateral speed need to be calculated.

As for the forward motion calculations, the aerodynamics that apply to a helicopter in flight also apply to one operating on the ground. Forward motion is caused by creating enough effective thrust, shown in Figure 3, to overcome the drag caused by the contact of the landing gear with the ground. To move forward then, power must be added and forward pitch must be applied.

Real helicopters have brakes to slow down and stop while on the ground. However, since the rudder pedal controls utilized for this research have no brakes equivalent, the effect must be simulated through other means. This was accomplished by programming pitch inputs from the joystick to control acceleration. To slow down or stop, the joystick must be pulled back and the throttle increased.

Heading control on the ground is the same as it is in the air, and that is by tail rotor control through rudder pedal input. Helicopter's do not have directly steerable wheels like automobiles do. Instead, the force produced by the tail rotor is quite sufficient to change the helicopter's heading since the torque produced by the main rotor, described in Chapter III, is still present. For this reason, the raw data from the rudder control pedals controls heading on the ground just as it does in a hover.

To protect the landing gear from damage, real helicopters are restricted from going above certain speeds while on the ground. The ground speed limit for the SH-60B is about 45 m.p.h., which is also the programmed limit for speed on the ground [NTPS90]. This gives the user adequate control and realism without subjecting the helicopter to extreme situations. These kinds of situations can cause serious damage to helicopters, and simulating this is beyond the scope of this research.

In addition, moving backwards on the ground is not permitted. Although most helicopters can taxi backwards easily, it is not practiced. There are two reasons for this. The first reason is that since helicopters don't have rear windshields like automobiles do, it is impossible for the pilot to see behind the helicopter to prevent colliding with anything. Secondly, to taxi backwards, the pilot must pull back on the cyclic and add power from the collective. Since doing this causes the main rotor system to pitch back, the possibility exists for the main rotor blades to impact the rear portion of the fuselage.

If the pilot flying this simulated helicopter lands while moving faster than the maximum forward ground limit or backwards, the helicopter is programmed to speed up or slow down gradually until ground speed is within limits. Flight can be achieved again, however, before the speed is in limits.

C. SUMMARY

The purpose of this portion of this research was to give helicopters in NPSNET the same controllability that real helicopters have. Due to the aerodynamic complexity of real helicopters, though, a completely physically based model was neither attempted nor desired. The purpose here is twofold. First, NPSNET must still be used by anyone, including those with no helicopter piloting experience, while giving the user some idea of what it takes to fly a real helicopter. The previous method of helicopter control gave users adequate control of a helicopter, but sacrificed too much realism to be a useful indicator of helicopter flight.

The second purpose behind this design is to execute realistic control of the helicopter without requiring special hardware to make complex calculations. Since NPSNET is designed to run on commercial off-the-shelf computer systems, the blade element method to calculate helicopter aerodynamics is too computationally expensive to be useful in NPSNET. The simpler balance of forces method provides enough realism to the simulation without sacrificing speed or usability.

VI. INTERFACE FOR MULTIPLE WORKSTATION ASW

A. INTERFACE PANEL DESIGN

As mentioned in Chapter II, the Panel interface designed by Chris McMahan provides a useful starting point to implement an ASW interface. This chapter describes the shortcomings of the original system and the changes made to the system for this research to provide ASW functionality.

B. COMMUNICATION WITH NPSNET

1. Original Implementation

Communications between the interface Panel and a workstation running NPSNET in remote mode are conducted through a single socket structure. This structure, called a GUI_MSG_DATA packet, is shown in Figure 18. This packet is used to pass control and weapons information between NPSNET and the Panel across the network.

The proper use of this packet is shown in Figure 19. In this case, one Panel is being used in conjunction with one workstation running NPSNET in remote mode. Remote mode means that the Panel displays velocity and position information about the NPSNET vehicle, herein called the host vehicle, and can be used to control that vehicle. This mode is turned on when NPSNET is initialized with the “-r” command line switch. This allows NPSNET and the Panel to communicate by passing the GUI_MSG_DATA packet over the network.

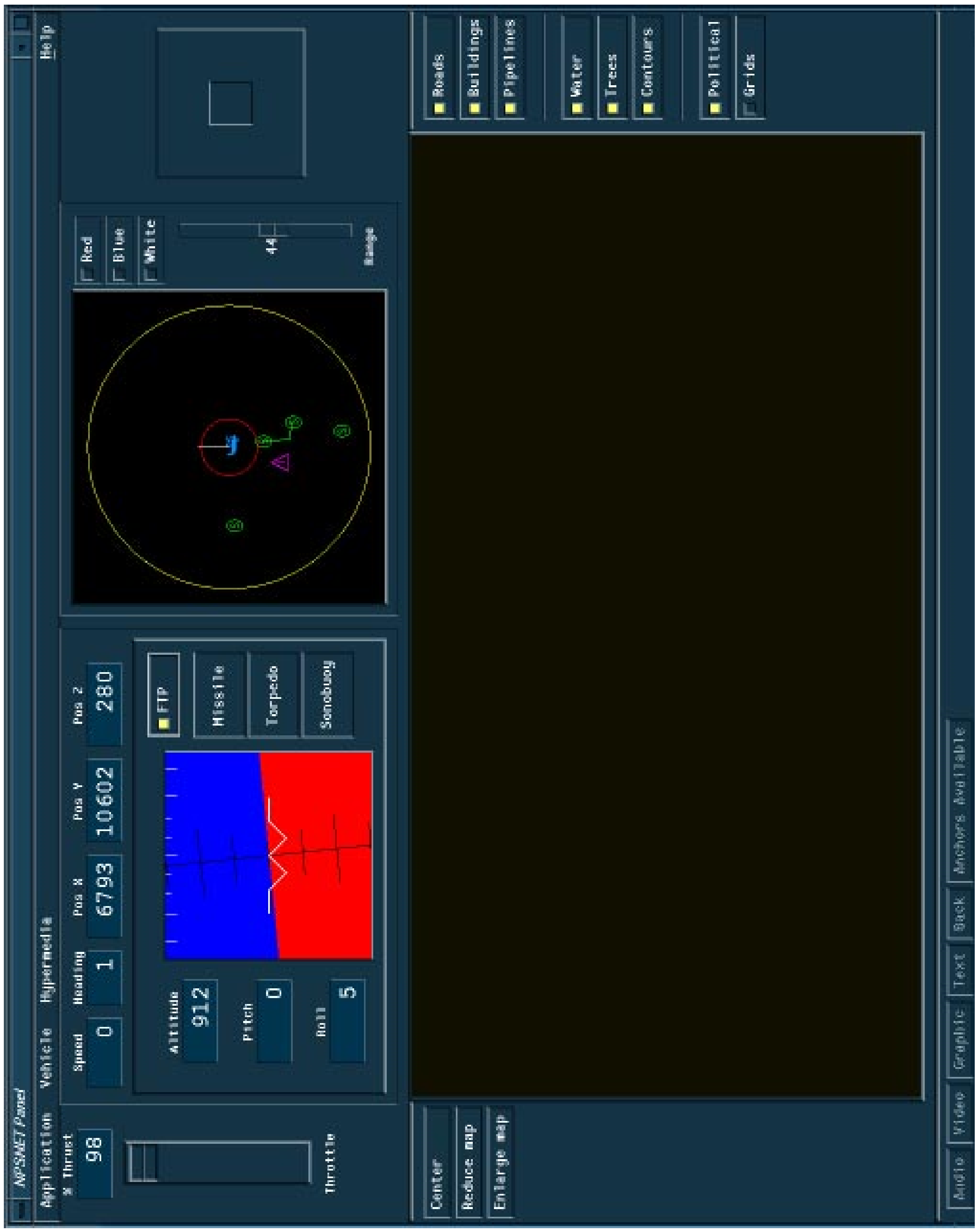


Figure 17: Interface Panel in ASW Helicopter Mode

```
typedef struct {

    double status;
    unsigned short type;
    unsigned short length;

    // throttle data required to read the throttle position (-1.0 to 1.0) and to
    // set the throttle input with the scale widget
    float throttleSetting;

    // joystick data required to read the joystick position and set input as required
    // (each is between -1.0 and 1.0)

    float joystickX;
    float joystickY;

    // vehicle settings read from NPSNET only (not sent from interface)
    float positionX;
    float positionY;
    float positionZ;

    float altitude;
    float heading;
    float pitch;
    float roll;
    float velocity;

    float gunAzimuth;
    float gunElevation;

    EntityID vehicleID;

    // flag settings to execute NPSNET actions
    EntityID targetVehicleID;
    BYTE attachMode;  // including tether, attach, target,teleport
    BYTE weaponsMode; // including primary, secondary,tertiary, targetingEnable
    BYTE hudMode;     // including hudEnable
    BYTE environmentMode;  // including fogEnable, wireframeEnable, textureEnable,
                           // cameraEnable

    // variable settings to control NPSNET functions
    BYTE fogSetting;
    BYTE hudSetting;

    //NEW - data passed to NPSNET about FTP use and position
    int FTPon;
    float FTPXposit;
    float FTPYposit;

} GUI_MSG_DATA;
```

Figure 18: GUI_MSG_DATA structure

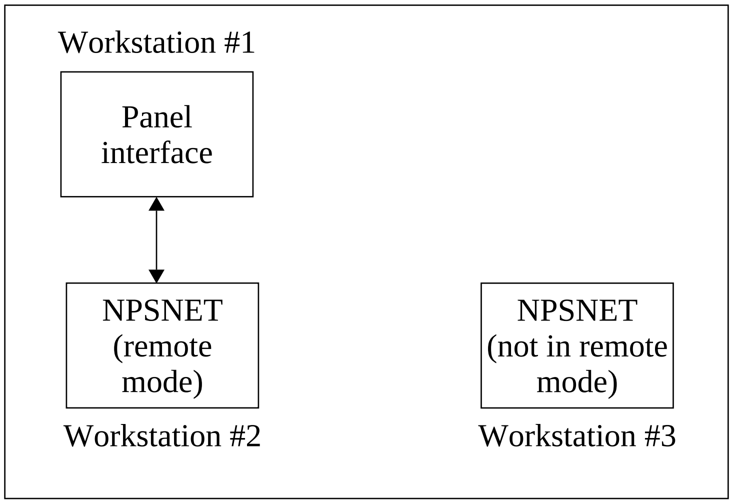


Figure 19: Single Panel/Single NPSNET Cooperation

The use of this method fails when more than one workstation is running NPSNET in remote mode, as shown in Figure 20. In this example, Workstations #2 and #3, each running NPSNET in remote mode, both communicate to the Panel application through the passing of GUI_MSG_DATA packets over the network. The Panel becomes unusable since conflicting information is received from different NPSNET host vehicles. In addition, if the Panel sends controlling information to NPSNET, it is possible for the host vehicles being simulated on Workstations #2 and #3 to be controlled by the one Panel on Workstation #1. This is generally not a realistic scenario: one person cannot drive two separate vehicles.

The situation becomes even more confusing when there are multiple interface Panels and multiple workstations running NPSNET in remote mode, as in Figure 21. Each panel could control both NPSNET host vehicles, and each Panel receives conflicting GUI_MSG_DATA information from both host vehicles.

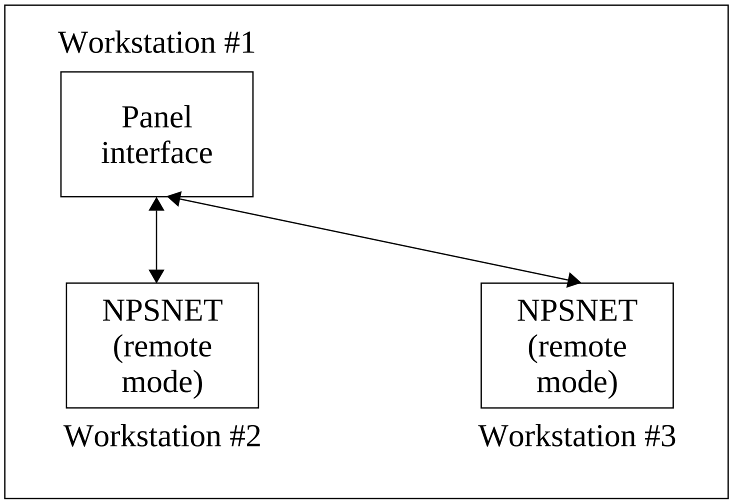


Figure 20: Single Panel/Multiple NPSNET Problem

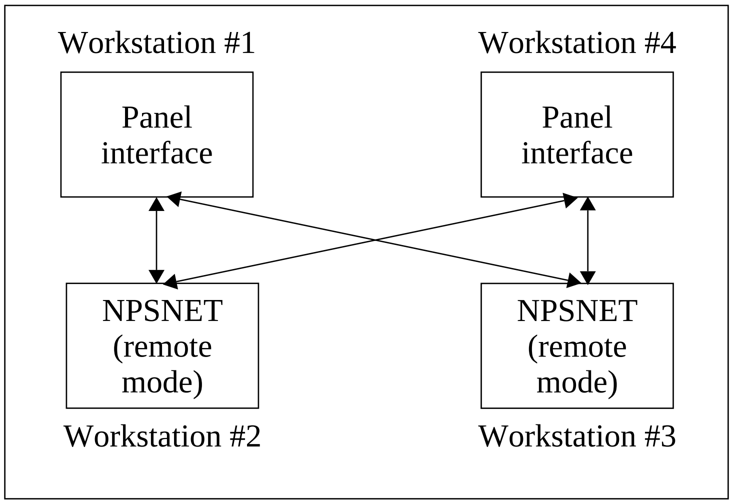


Figure 21: Multiple Panel/Multiple NPSNET Problem

2. Exclusive Two-way Communication with NPSNET

As described in Chapters III and IV, helicopter ASW is conducted through the utilization of each member of the helicopter's crew. Each crewmember performs specific functions while supplementing the duties of fellow crewmembers.

The functionality of the interface Panel provides an adequate framework to model helicopter ASW. However, to model the crew coordination concept, it is necessary that one Panel communicate with one NPSNET host vehicle, regardless of how many workstations are running NPSNET in remote mode.

The approach taken to accomplish this was to have one NPSNET workstation running in remote mode cooperate with one Panel, as shown in Figure 22. Each workstation supplies address information to the GUI_MSG_DATA packets it generates. When either workstation receives the packets, it compares the address information in the packet to the address information stored in the program about which workstation it is cooperating with. If the addresses match, the state of the respective program is updated. Otherwise, the GUI_MSG_DATA packets are discarded.

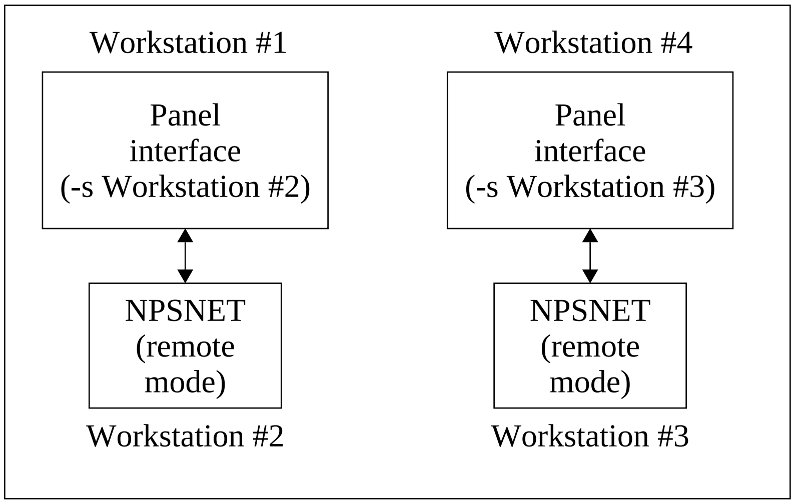


Figure 22: Desired Multiple Panel/Multiple NPSNET Cooperation

a. Communication from the Interface Panel to NPSNET

The first step to accomplish this was to change how the interface panel is initialized so that a specific workstation could be designated as the cooperative workstation. This designated workstation will run NPSNET in remote mode, and it is the only workstation which the Panel will communicate with. A new command line initialization switch was implemented to supply the necessary workstation identification information to the Panel at start-up.

To initialize the Panel in this mode, the new switch uses the character ‘s’ to designate a session. The syntax of the switch is “-s <workstation name>”. When the name of a valid workstation is supplied, the identification information allows both NPSNET and the Panel to filter out unwanted GUI_MSG_DATA packets from other workstations. The workstation name is read from the command line at initialization and then verified in the *datafiles/hosts.dat* file, an example of which is shown in Figure 23. The two pieces of applicable information are the second and third fields of the file, indicating the site number (all 36 in the example) and the host number (the number following 36). This information is stored in the variable “npsnetAddress.address.site” and “npsnetAddress.address.host”, respectively, as part of the GLOBAL_DATA structure defined in *global.h*.

After initialization of the Panel in complete, it can now communicate with the desired workstation running NPSNET in remote mode by passing the GUI_MSG_DATA packet. When a change is made to the Panel’s state that requires a similar change of state in NPSNET, the applicable data is sent to the Message class of the Panel [MCMA95]. The Message class then generates a GUI_MSG_DATA packet, writing the applicable information to the appropriate field of the packet. The last step performed by the Message class is to write the “npsnetAddress.address” data field to the “vehicleID” field of the GUI_MSG_DATA packet and write the packet to the network. This is performed in Message::writeSocket(), seen in Appendix B beginning on line 5.

```
bessie     36   1  ec0
betsie     36   2  ec0
black      36   3  ec0
bond       36   4  ec0
bossie     36   5  ec0
bradley    36   6  ec0
clancy     36   7  ec0
clarabell  36   8  ec0
cool       36   9  ec0
daisy      36  10  ec0
dude       36  11  ec0
elsie      36  12  ec0
elvis      36  13  et0
fletch     36  14  ec0
gnarly     36  15  ec0
grant      36  16  ec0
gravy1     36  17  ec0
gravy2     36  18  ec0
gravy3     36  19  et0
gravy5     36  20  et0
holmes     36  21  ec0
like       36  22  ec0
meatloaf   36  23  et0
medusa     36  24  ec0
minotaur   36  25  ec0
patton     36  26  ec0
pershing   36  27  ec0
rambo      36  28  ec0
ridgeway   36  29  ec0
totally    36  30  ec0
trouble    36  31  ec0
turing     36  32  ec0
chaos   36  33  ec0
federation.arl.mil 36  34  et0
orion.arl.mil 36 35 et0
hred-iport3.arl.mil 36  36 et0
hred-iport2.arl.mil 36 37 ec0
```

Figure 23: Example *hosts.dat* File

When any NPSNET workstation running in remote mode receives this packet, it first compares the “vehicleID” field in the packet with its own identification information, which is obtained from the same *hosts.dat* file used by the Panel. This is performed in the main application loop of NPSNET in the source file *main.cc*, shown in Appendix A beginning on line 9. If there is a GUI_MSG_DATA packet available to be read, and if the “vehicleID.address” field of the packet matches the workstation’s own

"DIS_address", the rest of the information in the packet is assigned to the appropriate place to update the NPSNET host vehicle's state. If the GUI_MSG_DATA "vehicleID.address" field does not match the workstation's own address information, the packet is discarded.

The situation described here is initiated when the Panel has remote control of the NPSNET host vehicle and any control input changes are made. This includes weapons fire information and the newly implemented Fly-to-Point data, both described later in this chapter. This filtering of GUI_MSG_DATA packets now allows one Panel to cooperate with a host vehicle being simulated on one workstation, while other Panels communicate with their own NPSNET host vehicles. This operates as shown in Figure 22.

b. Communication from NPSNET to the Interface Panel

The previous section described the situation where the interface Panel sends control information to NPSNET running in remote mode. NPSNET also sends the GUI_MSG_DATA packet to the Panel when the host vehicle changes its state. This is necessary when the host vehicle changes any data that gets repeated on the Panel's display, including heading, pitch, roll, position, velocity, and altitude.

To maintain the capability shown in Figure 22, the Panel must also filter out any GUI_MSG_DATA packets it receives from unwanted NPSNET workstations running in remote mode. This is done in a similar manner as when the Panel generates the packets for NPSNET, with the exception that NPSNET puts its own address information into the packet's "vehicleID.address" field.

In each cycle of NPSNET's main application loop, every field of the GUI_MSG_DATA packet that involves control data is updated with the host vehicle's appropriate information, shown in Appendix A starting on line 82. As can be seen on lines 94 an 95, the workstation's own address information is written to the "vehicleID.address" field of the GUI_MSG_DATA packet. After five cycles of the main application loop, the latest packet to be updated is sent to the network.

When the Panel receives a GUI_MSG_DATA packet from NPSNET, it first compares the "vehicleID.address" field of the packet with the "nspnetAddress.address" variable assigned upon initialization and mentioned in the previous section. This is shown in the readSocket() function beginning on line 26 of Appendix B. If the address information matches, the packet was sent from the cooperating workstation that was designated with the "-s" switch. Therefore, the packet is sent to the appropriate Panel classes to update any displayed data, just as had been done in the Panel's original implementation [MCMA95]. However, if the addresses differ, the packet is discarded and the Panel's display is not updated.

With this, the two-way exclusive communication that was the goal of this research and illustrated in Figure 22 has been achieved. Both NPSNET and the Panel get updated by receipt of GUI_MSG_DATA packets addressed to or from the desired workstation while all others are discarded.

C. GUI_MSG_DATA PACKET UTILIZATION FOR ASW

Two of the duties listed in Chapter IV required by the ATO in an ASW helicopter are to navigate to target areas for employment of sensors and weapons, and to direct sonobuoy and weapon delivery. These duties require communication between the pilot, who is flying the helicopter, and the ATO. This crew coordination must be conducted efficiently to adequately prosecute a submarine threat.

With the two-way exclusive communication implemented between the interface Panel and NPSNET, the crew coordination concept can now be modelled. By incorporating navigation and ASW weapons delivery functionality into the Panel, the functions performed by the ATO are simulated. The data generated by the ATO is transmitted to the selected NPSNET workstation operating in remote mode to simulate crew coordination. Host vehicle data is also transmitted from the pilot's workstation back to the ATO's Panel so the ATO can back up the pilot in maintaining safety of flight. All of this is performed through the use of the GUI_MSG_DATA packet.

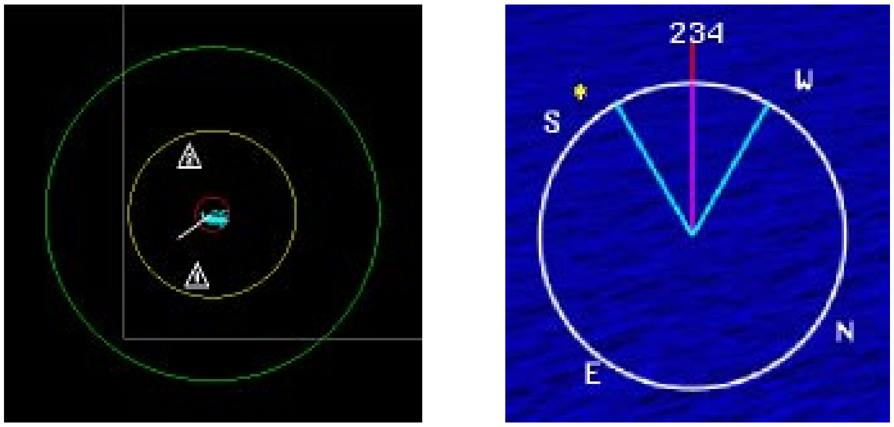
1. Fly-to-Point Navigation

The way the ATO informs the pilot where to fly the helicopter is through the use of fly-to-points (FTP). During an ASW mission, the ATO decides where to deploy sonobuoys and launch weapons. The location he decides on is a precise location based on the tactics used to prosecute the submarine. The ATO enters FTP information onto his tactical display as a prioritized list of locations. The highest priority FTP is the one the pilot will fly to next.

The information regarding this FTP is displayed to the pilot as a heading cursor on his compass. The pilot turns the helicopter until its heading and the heading to the cursor are the same. Once the helicopter reaches this FTP, the point disappears, which is called capturing the FTP. The ATO can then launch a sonobuoy or weapons as he desires. The pilot also gets information about the FTP with the next highest priority after capturing the current one.

To model this functionality, the ability of the ATO to designate FTP's on the Panel and communicate their location to the pilot's NPSNET workstation was implemented. This is shown in Figure 24. The left side of the figure shows the radar display of the Panel, which is used as the ATO's tactical display. It shows the helicopter in the center, with a line from its center pointing down and to the left to indicate the helicopter's heading. The two triangles are FTP's, with the one below the helicopter having highest priority and the one above having a lower priority. The right side of the figure shows the compass on the heads-up display (HUD) for this same helicopter in NPSNET. The '*' next to the 'S' indicates to the pilot which way to head to capture the FTP, in this case, to turn left from heading 234. How the ATO enters FTP information onto the display is shown in the next chapter.

When the ATO enters an FTP onto the Panel's radar, the function Message::FTPChanged() is called, which generates a GUI_MSG_DATA packet to notify the pilot where to go. The FTP location information is written in world coordinates to the "FTPxposit" and "FTPyposit" fields of the GUI_MSG_DATA packet, seen at the bottom of Figure 18. The packet is then sent to NPSNET over the network as described earlier.



a) Panel radar display b) NPSNET HUD display

Figure 24: Communication of FTP Information

When the pilot's NPSNET workstation receives the packet, the FTP position data is written to the FTP data fields of the PASS_DATA structure defined in *draw.h* in the source code of NPSNET. This assignment is shown on lines 39 through 41 in Appendix A. This data is then used by the vehicle_hud_draw() function in NPSNET's *vehicle.cc* to display the heading cursor on the pilot's compass. This data is displayed until the FTP is captured or the ATO deletes this FTP or designates a new one on the radar display of the Panel.

2. Weapons Delivery

Once the ATO has communicated to the pilot where to fly the helicopter, it is up to the ATO to launch weapons and sonobuoys, as described earlier. Since weapons in NPSNET are themselves entities, generation of the weapon is handled by the vehicle's workstation, which sends DIS entity state, fire, and detonation PDU's regarding the weapon to the other vehicle's participating in the simulation.

To model this capability, use of the GUI_MSG_DATA packet is again needed. When the ATO generates a torpedo or missile launch from the weapons buttons on the Panel, shown in Figure 17, a GUI_MSG_DATA packet is generated by calling the weapon launch functions of the Message class. These are shown in Appendix B beginning on line 155. The data sent to NPSNET is in the form of a byte called “weaponsMode” in the GUI_MSG_DATA packet, as seen in Figure 18. It specifies whether to launch the vehicle’s primary, secondary or tertiary weapon. The packet is then sent onto the network.

When the NPSNET workstation meant to get the packet receives it, the “weaponsMode” byte value is stored in a variable called “the_button”, seen on line 21 of Appendix A. If the vehicle type is an ASW helicopter, then “the_button” is compared with constants defining primary, secondary, or tertiary weapons. Whichever value matches, a firebuttonhit() function call is made to the Munitions class in NPSNET, and the appropriate weapon is fired just as if the person at the flight controls launched it.

Since the ATO generally launches the torpedoes or missiles in an SH-60B helicopter, the capability was given to have the ATO running the Panel be able to launch weapons even if the pilot operating the vehicle in NPSNET has flight control over the host vehicle. This simulates additional crew coordination functionality between the pilot and ATO. The pilot also still has the ability to launch weapons from buttons on the flight controls or from the keyboard.

D. SUMMARY

The communications conducted between the interface Panel and NPSNET prior to this research contained adequate framework to simulate crew coordination. By programming both the Panel and NPSNET to filter out unwanted GUI_MSG_DATA packets, a single Panel can now communicate with a single NPSNET entity while other Panels operate in conjunction with their entities. These crew coordination concepts are shown through the added capability of passing FTP data from the Panel to the NPSNET entity, and through the flight data passed from the entity to the Panel for display.

VII. ASW SYSTEM DESCRIPTION

A. INTRODUCTION

This chapter describes the ASW functionality added to NPSNET and the interface Panel, including a description of the radar display, how to enter and manipulate Fly-to-Points (FTP), how sonobuoys are modelled, and how to launch torpedoes and missiles. The ASW functionality is available only when NPSNET and the interface Panel are initiated in the correct mode. NPSNET must be initialized as an SH-60B helicopter in remote mode as the host vehicle. This is accomplished by including the “-v” command line switch with “sh60” as its argument. Remote mode is turned on by including the “-r” switch at initialization, which allows NPSNET to send and receive GUI_MSG_DATA packets.

To activate the ASW functionality in the interface Panel, it must be initialized in session mode with the “-s” command line switch with the name of the host vehicle’s workstation as its argument, as described in Chapter VI. For example, if we want to control the NPSNET host vehicle on the workstation “elvis”, initialized as described above, then the Panel must be initialized from another workstation with the argument “-s elvis”. This allows the Panel to send and receive GUI_MSG_DATA packets, and enables packet filtering for both NPSNET and the Panel, also described in Chapter VI. Other ASW functionality is activated by selecting “ASW Helo” from the Vehicle menu of the Panel. This enables FTP and sonobuoy functionality. Figure 17 shows the Panel in ASW Helicopter mode.

B. PANEL RADAR DISPLAY

The ATO in an SH-60B has a tactical display used for viewing sensor data, managing sonobuoys, entering FTP’s, and controlling the tactical conduct of a mission. This display, shown in Figure 6, is graphical in nature, giving the ATO a picture of the ASW situation.

The SH-60B tactical display is simulated in this research through the use of the radar display on the interface Panel. The basic functionality from the original implementation of the Panel remains largely unchanged [MCMA95]. Its function is to display the location of entities in the virtual environment relative to the host vehicle's location. The center of the radar display represents the location of the host vehicle. All other entities, whose locations are determined from DIS Entity State PDU's received from the network, are shown on the radar as icons representing the vehicles' type.

One significant change made to the radar display of the Panel was its orientation. Originally, the heading of the host vehicle determined the radar's orientation, with the vehicle's heading pointing up. So, the radar picture rotated as the host vehicle turned. This was an unrealistic depiction of a radar's display, and can get confusing for the user. Real radar displays are oriented such that north points up on the radar's display. When the vehicle turns, the display stays oriented towards north and the iconic representation of the vehicle turns. The icon indicates the heading of the helicopter with a small line inside of the icon.

To simulate this, the rotating functionality of the Panel's radar display was removed. As a result, north points up on the display regardless of vehicle heading. To indicate the host vehicle's heading, a small line is drawn from the center of the radar pointing in the direction of the host vehicle's heading. This can be seen in Figure 25, which shows that the host vehicle's heading is about southwest.

Additional functionality was added to the Panel's radar display to allow for the display and management of FTP's and sonobuoys. The source code for the Radar class of the interface Panel is contained in Appendix C and Appendix D.

1. Fly-to-Point Management

FTP's are navigational aids used by the ATO to tell the pilot where to fly the helicopter. They are used in ASW to indicate locations for sonobuoy and weapons deployment.

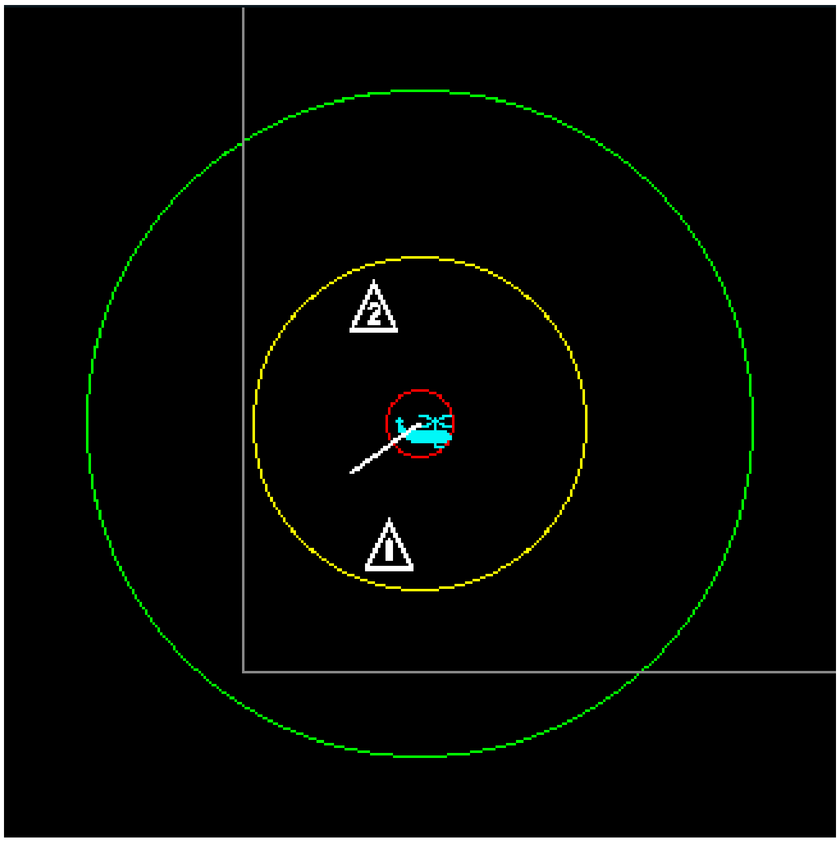


Figure 25: FTP Example

In the interface Panel, FTP's are entered and displayed in the radar display, as shown in Figure 25. Each FTP is displayed as a triangle with a number inside indicating its priority. Up to six FTP's can be entered on the radar display. Source code for displaying, entering, deleting, and capturing FTP's is contained in Appendix C and Appendix D.

a. Entering FTP's

Entering FTP's onto the radar display is made possible by turning on the FTP toggle button, as shown in Figure 17. New FTP's appear on the radar display at the mouse pointer's position when the left mouse button is pressed in the radar's display area. If there are a maximum number of FTP's in the system, in this case six, no more can be entered until one of the existing FTP's has been deleted or captured.

As mentioned earlier, each FTP also has an associated priority, which indicates the order in which the FTP's should be flown to. Newly entered FTP's have the lowest priority of those in the system. For example, if there are three FTP's currently entered into the system, the next FTP will have priority 4, assuming that none of the original FTP's gets deleted from the system before the new one is entered. If there are no FTP's currently in the system, the new FTP will have priority 1.

The significance of the highest priority FTP (priority 1) is that it is the position selected by the ATO for the next sonobuoy or weapon launch. As such, it is the only FTP whose position is transmitted to the host vehicle, which is described in Chapter VI.

b. Capturing FTP's

When the helicopter gets within a certain distance of the highest priority FTP, it is deleted from the system, signifying that the position the ATO wanted the pilot to fly to has been reached. This distance is calculated as a horizontal distance since FTP's are not based on altitude. The maximum distance this can occur is defined at compile time to be 500 yards. Hence, when the helicopter gets within 500 yards of the priority 1 FTP, it is captured. Only the priority 1 FTP can be captured.

After capturing the priority 1 FTP, the priorities of any other FTP's in the system are increase by one. So, if the priority 1 FTP in Figure 25 is captured, it is removed from the radar display, and the priority of the priority 2 FTP becomes priority 1, whose location is then transmitted to the NPSNET host vehicle. In addition, if there are six FTP's in the system prior to one being deleted or captured, then there are five FTP's after. In this case, another FTP can then be added to the system, as described earlier.

As stated before, FTP's denote the locations where the ATO wishes to drop sonobuoys or torpedoes. One of the capabilities of the SH-60B is that it can automatically launch sonobuoys upon the capture of an FTP. To simulate this, the Panel automatically inserts a sonobuoy into the system when capturing an FTP. Sonobuoy use is described later in this chapter.

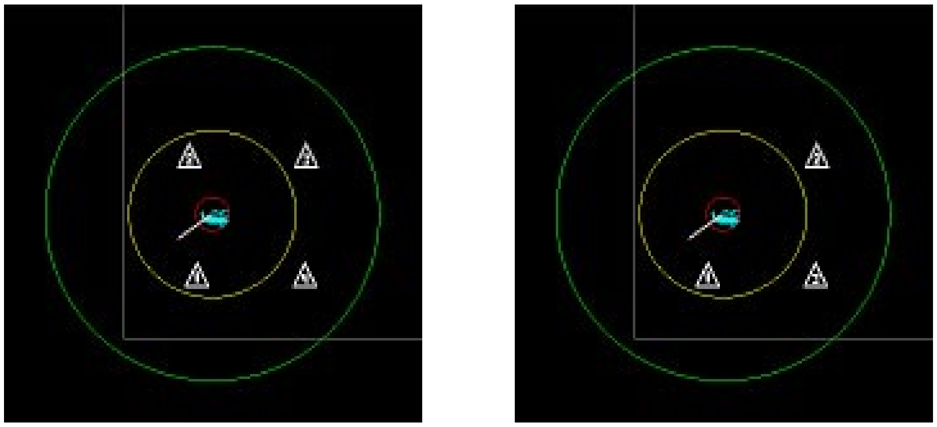
c. Deleting FTP’s

Occasionally, the ATO may wish to delete an FTP from the system. This is done when the ATO wishes to reprioritize them, change their locations, or insert new ones if the maximum number is already in the system.

Deleting FTP’s is accomplished on the Panel by first turning the FTP toggle button off. Then, by placing the mouse pointer on a currently existing FTP and clicking the left mouse button, the FTP disappears from the radar display. In addition, remaining FTP’s are reprioritized, if necessary, so that the priorities of all FTP’s in the system is an increasing sequence (i.e. no priority gaps). This is shown in Figure 26. On the left side, there are four FTP’s with priorities 1 though 4. When the FTP with priority 2 is deleted, the rest of the FTP’s get renumbered such that the FTP with priority 3 now has priority 2, and the FTP with priority 4 now has priority 3.

2. Sonobuoy Usage

As described in Chapter V, sonobuoys are used to track subsurface contacts. For this research, sonobuoys are modelled only as part of the interface Panel. The reason for this is that, as a sensor, the sonobuoy would need to be modelled as a sensor entity in NPSNET to provide the necessary information to other vehicles in NPSNET regarding its characteristics. This would necessitate the use of the DIS Emission PDU, currently unimplemented in NPSNET.



a) before deleting FTP b) #2 FTP deleted

Figure 26: Deleting FTP’s

To model sonobuoys through the Panel, the weapons launch buttons and the radar display are again used. This allows for entering sonobuoys into the system, displaying sonobuoy locations, and displaying subsurface contact information. Sonobuoys are displayed as a circle around the letter ‘S’ with a line indicating the submarine’s position, as shown in Figure 27.

a. Entering Sonobuoys

Sonobuoys can be entered in to the system automatically or manually. Since sonobuoys are launched from the helicopter’s current location, sonobuoys first appear in that location. The sonobuoy remains in that location for the duration of the simulation.

To automatically enter a sonobuoy into the system, the helicopter must capture a FTP, as described earlier. Manually launching sonobuoys is accomplished by pressing the sonobuoy button next to the radar display. A maximum of 25 sonobuoys can be entered into the system. There is no difference in the functionality of manually launched sonobuoys versus those automatically launched.

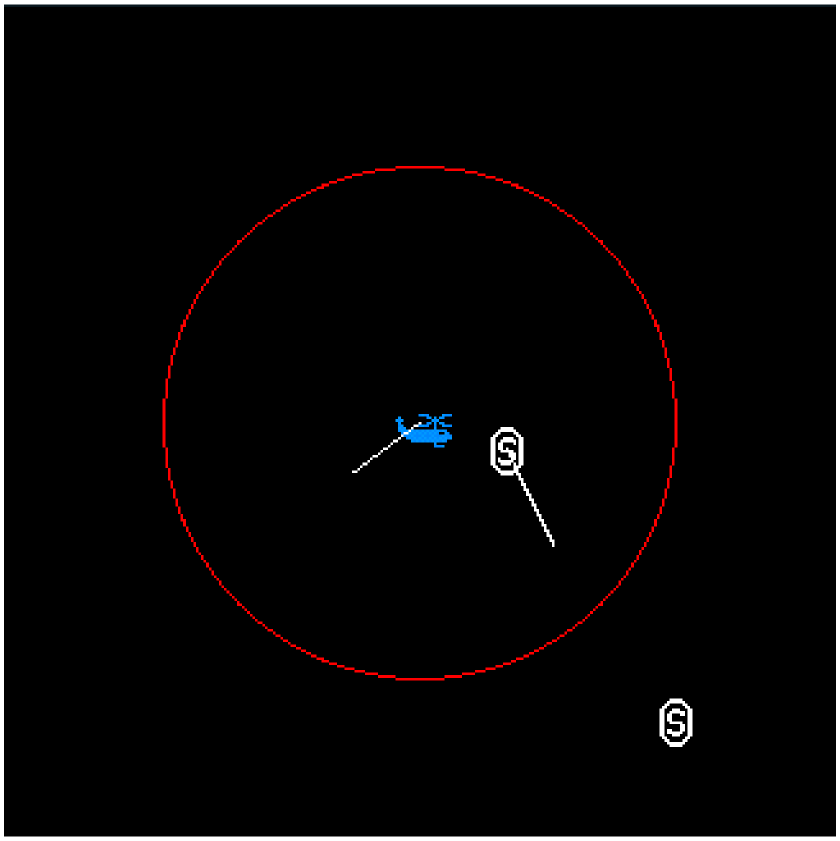


Figure 27: Sonobuoy Examples

b. Sonobuoy Contact Information

The type of sonobuoy modelled in the interface Panel is an active sonobuoy. As described in Chapter IV, active sonobuoys reveal bearing and distance information from the sonobuoy to the subsurface contact by transmitting a sonar signal and receiving the sonar echo. Due to their limited power, both active and passive sonobuoys are generally considered short range sensors for tracking submarines.

Sonobuoys simulated in the Panel do not detect subsurface contacts by listening for acoustic signals. This is because entities in NPSNET transmit DIS Entity State PDU’s, which have no acoustic information. The Panel, however, receives these PDU’s to be able to display these entities on the radar display. By extracting vehicle type and position information from the PDU’s, the Panel system knows the location of all vehicles in the simulation, including submarines.

To display the location of a submarine relative to a sonobuoy, the distance from the sonobuoy to the submarine is tested. To simulate the limited range of sonobuoys, the location of a submarine is displayed only if its distance is less than 1000 yards. If it is, a line segment is drawn from the sonobuoy to the submarine’s location. If the distance is greater than 1000 yards, no line segment is drawn. Both of these situations are shown in Figure 27, where the top sonobuoy is in contact with the submarine, but the bottom one is not. In addition, the iconic representation of the submarine is not shown. This is omitted to simulate a radar’s inability to detect subsurface vehicles and to offer the greater challenge of locating the submarine without knowing any position information when the simulation is begun.

Multiple sonobuoys can indicate contact with a single submarine. If one submarine is within 1000 yards of two or more sonobuoys, lines are drawn from each sonobuoy to the submarine, as shown in Figure 28. If there is more than one submarine within range of a sonobuoy, only the position of the closest submarine is indicated on the radar display.

3. Weapons Launch

Weapons can be launched in NPSNET either by the person controlling the host vehicle in NPSNET or by the person operating the interface Panel. To launch weapons in NPSNET, the pilot can either press the appropriate button on the joystick, or press the appropriate button on the keyboard. This functionality was not changed for this research.

Changes were made to weapons launch functionality in the interface Panel, however. To launch weapons from the Panel, the buttons labelled “Torpedo” and “Missile”, shown in Figure 17, are used. At any point in the simulation exercise, the person operating the Panel places the mouse pointer on the appropriate weapon button and presses the left mouse button. This sends a GUI_MSG_DATA packet to the NPSNET host vehicle, which launches the selected weapon. Since the ATO has the responsibility of launching weapons in the SH-60B, the Panel can be used to launch weapons regardless of who has control of the host vehicle in NPSNET, either the person flying the NPSNET helicopter or the person controlling the Panel.

C. WEAPONS SIMULATION

The SH-60B has the capability to carry and launch Penguin missiles and Mk 48 or Mk 50 torpedoes. This is simulated in a general sense in NPSNET. The SH-60B in NPSNET has two weapons, a missile and a torpedo. The missile, representing the Penguin, is a generic missile that can home in on a targeted vehicle. No functionality was added to the current simulation of missiles in NPSNET in this research.

To simulate torpedoes, a new munitions class was added to NPSNET to represent helicopter launched torpedoes. What is unique about this kind of weapon in NPSNET is that is has two separate trajectory calculations, one for the airborne ballistic phase, and one for the underwater homing phase. In addition, the torpedo only has the capability to home in on and destroy ships and submarines in NPSNET.

When a torpedo is launched, an array is created containing the identification numbers of vehicle in the current simulation that are eligible to be targeted, which are ships and submarines. This is done to allow a quick lookup for the torpedo when trying to gain a contact to start homing in on the target. The torpedo itself follows a ballistic trajectory until water impact.

Once the torpedo impacts the water, the homing phase begins. If there are any eligible targets in the simulation, the distance from the water impact point to the vehicle is calculated. If any of these vehicles are within the torpedo’s acquisition range, defined at compile time to be 3000 yards, the closest of these vehicles becomes the target. The torpedo then turns toward the target. If there are no eligible vehicles in range, the current distance from the torpedo to the eligible vehicles is checked on each display loop in case a vehicle enters the torpedo’s acquisition range before the end of its run. If no vehicles are acquired,

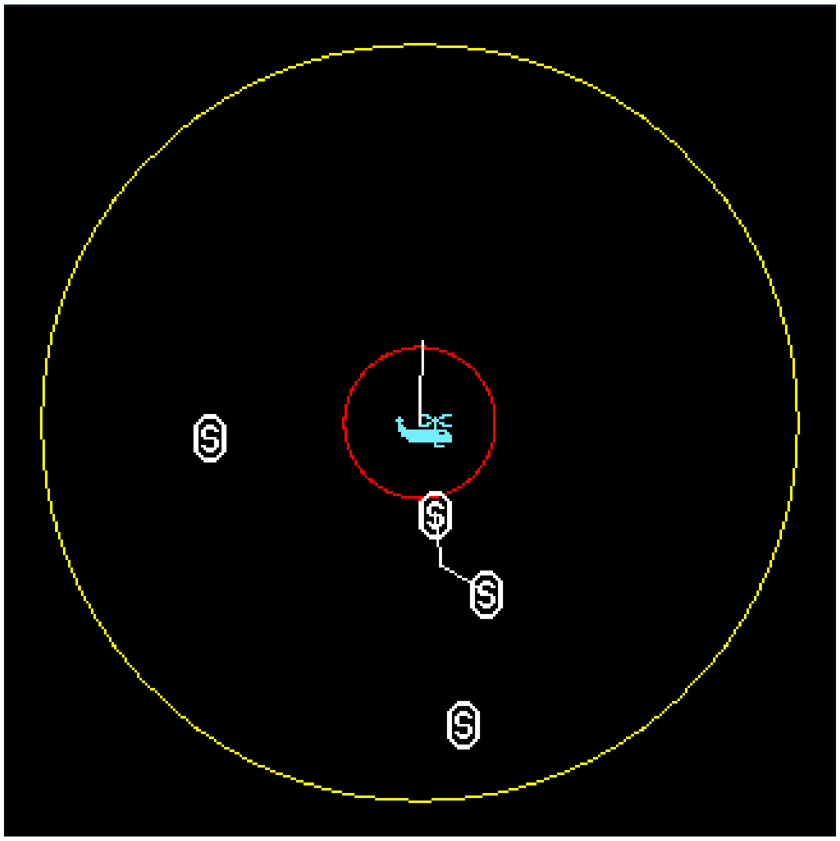


Figure 28: Multiple Sonobuoy Contact

the torpedo continues on it current heading, maintaining its last calculated pitch and roll angles.

In addition to homing toward the closest ship or submarine, calculation of the distance the torpedo has travelled since water impact is made on each display loop. This is done to limit the range of the torpedo. If it has run for 2000 yards underwater without hitting a target, the torpedo is deactivated.

D. SUMMARY

By adding ASW functionality to NPSNET and the interface Panel, crew coordination concepts as they apply to ASW can now be practiced in simulations over a network. The features added to simulate ASW, described in this chapter, provide to users the ability to prosecute subsurface entities in a realistic training environment. The interface

Panel, with the changes incorporated as a result of this research, now provides the functionality to realistically simulate the warfare area of ASW.

VIII. RESULTS, CONCLUSIONS, AND FUTURE RESEARCH

A. RESULTS

The result of this research is a simulation environment that incorporates real world concepts in ASW warfare. From the standpoint of the helicopter aircrew, realism was added to every aspect of the simulation to allow for a more useful training environment.

The first step was to increase the realism for the pilot flying the ASW helicopter. A full set of flight controls was implemented to allow the pilot to have full control over the helicopter. The model of the helicopter was changed from a simple one vector representation to a more complex four vector representation. This model uses Performer to manipulate the vectors, so no speed is sacrificed with the increase in complexity of the model. This gives the helicopter the ability to hover, slide left or right, and fly backwards and forwards. The realism for the pilot is that he must maintain safety of flight by concentrating on his flying technique, letting the other members of the aircrew handle their own responsibilities.

Next, the interface Panel was adjusted to add realism for the ATO. By having the Panel communicate with the host NPSNET vehicle through the GUI_MSG_DATA packet, the ATO can communicate to the pilot his intentions for the conduct of the mission. The pilot knows where he must fly, and once that point is reached, the ATO can deploy his sensors and weapons as necessary to prosecute the submarine.

To allow for realistic joint training to occur, GUI_MSG_DATA packet filtering by both NPSNET and the Panel is used. This allows each Panel to communicate with its own NPSNET host vehicle. Several aircrews can prosecute submarines in a simulation, which is how real world helicopter ASW would be conducted.

B. CONCLUSIONS

The objectives of this research were to design and construct an interface to simulate helicopter ASW and to increase the realism of helicopter control in NPSNET. The first objective was accomplished by adding ASW functionality to the interface Panel and NPSNET. The second objective was accomplished by redesigning the helicopter control model previously implemented in NPSNET. The functionality was developed and successfully incorporated into NPSNET and the interface Panel. The following conclusions have been reached.

- Complex control models can be incorporated into NPSNET which provide a higher degree of realism for training pilots from a variety of airborne platforms.
- Joint training and crew coordination concepts can be simulated through the use of a common interface panel representing a variety of vehicle types and warfare areas. Communication between the interface and the host vehicle can be isolated so that they communicate only with each other.
- Simulation of specific warfare areas is possible by designing weapons and functions specific to the tactics used in that warfare area. Incorporating these into the simulation system allows for realistic warfare training.

C. FUTURE RESEARCH

This research provides a basic foundation for continued exploration of incorporating increased realism in NPSNET. The following is a list of topics for future research.

- Continued development and refinement of physically-based vehicle control models in NPSNET.
- Develop electromagnetic and acoustic sensor models and incorporate them into NPSNET.
- Incorporate acoustic properties into ship and submarine models of NPSNET.
- Further optimization of the network communications mechanisms to enhance speed and efficiency.
- Design and implement a larger variety of warfare area specific functions for the Panel and NPSNET.

APPENDIX A. SOURCE CODE FOR NPSNET IN REMOTE MODE (MAIN.CC)

```
#ifdef REMOTE_PANEL
    if ( G_static_data->remote_exists )
    {

     if ( ( G_remote = socket_read ( &rstat ) ) != NULL )
     {

       //NEW - Update the control settings only if the GUI_MSG_DATA is to us
       if (( G_dynamic_data->control == REMOTE ) &&
          (G_remote->vehicleID.address.site == G_static_data->DIS_address.site) &&
          (G_remote->vehicleID.address.host == G_static_data->DIS_address.host))
       {
         the_thrust = G_remote->throttleSetting;
         the_joy_y = G_remote->joystickY;
         the_joy_x = G_remote->joystickX;
         the_gunElevation = G_remote->gunElevation;
         the_gunAzimuth = G_remote->gunAzimuth;
       }

       // Update weapons delivery and fog modes
       the_button = G_remote->weaponsMode;
       the_fog = G_remote->environmentMode;

       //NEW - weapons fire and FTP info
       if ((G_remote->vehicleID.address.site == G_static_data->DIS_address.site) &&
          (G_remote->vehicleID.address.host == G_static_data->DIS_address.host))
       {
        // If the helo an SH60, let the panel fire weapons,  regardless of who has control
         if (isaswheloveh(draw_data->type))
         {
           if ( the_button & GUI_PRIMARY )
             firebuttonhit ( PRIMARY, G_drivenveh, draw_data);

           if ( the_button & GUI_SECONDARY )
             firebuttonhit ( SECONDARY, G_drivenveh, draw_data);
         } // end if ASW helo

       //NEW - get FTP data for display to user
       draw_data->FTP_is_on = G_remote->FTPon;
       draw_data->FTP_X_posit = G_remote->FTPXposit;
```

```
            draw_data->FTP_Y_posit = G_remote->FTPYposit;
            } // end weps and FTP info

        } // end there was a socket to read

        // If the Panel has control, update the vehicle's control information
        if ( G_dynamic_data->control == REMOTE )
        {
            control_data.thrust = the_thrust;
            control_data.pitch = the_joy_y;
            control_data.roll = the_joy_x;
            draw_data->look.hpr[HEADING] = the_gunAzimuth;
            draw_data->look.hpr[PITCH] = the_gunElevation;

            // Fire weapon if Panel has control
            if ( the_button & GUI_PRIMARY  )
                firebuttonhit ( PRIMARY, G_drivenveh, draw_data);

            if ( the_button & GUI_SECONDARY )
                firebuttonhit ( SECONDARY, G_drivenveh, draw_data);

            if ( the_button & GUI_TERTIARY )
                firebuttonhit ( TERTIARY, G_drivenveh, draw_data);

            // If Panel in control, toggle targetting
            if ( (the_button & GUI_TARGETING) != last_state )
            {
                G_dynamic_data->targetting = !G_dynamic_data->targetting;
                target_ok = G_curtime;
                last_state = (the_button & GUI_TARGETING);
            }

            // If panel in control, toggle fog
            if ( ( the_fog & GUI_FOG ) != last_fog )
            {
                changed.fog = TRUE;
                last_fog = ( the_fog & GUI_FOG );
            }
        } // end if control == REMOTE

        // Generate a GUI_MSG_DATA packet to send state info to Panel
        if ( sock_open )
        {
            static int loop = 0;
            info_to_go.heading = (float) (360.0f - (draw_data->posture.hpr[HEADING]));
            info_to_go.pitch = (float) draw_data->posture.hpr[PITCH];
            info_to_go.roll = (float) draw_data->posture.hpr[ROLL];
        info_to_go.throttleSetting = (float)control_data.thrust;
        info_to_go.gunAzimuth = (float) draw_data->look.hpr[HEADING];
        info_to_go.gunElevation = (float) draw_data->look.hpr[PITCH];
        info_to_go.positionX = (float) draw_data->posture.xyz[X];
        info_to_go.positionY = (float) draw_data->posture.xyz[Y];
        info_to_go.positionZ = (float) draw_data->posture.xyz[Z];
        info_to_go.vehicleID.address.site = G_static_data->DIS_address.site;
        info_to_go.vehicleID.address.host = G_static_data->DIS_address.host;
        info_to_go.altitude = (draw_data->altitude) * METER_TO_FOOT;
        info_to_go.velocity = draw_data->speed;

        //NEW - info to send to panel about FTP (not really needed)
        info_to_go.FTPon = the_FTPOn;
        info_to_go.FTPXposit = the_FTPXposit;
        info_to_go.FTPYposit = the_FTPYposit;

        loop++;

        // Write the socket to the network every application cycles
        if ( loop >= 5 )
        {
          socket_write ( (char *) &info_to_go );
          loop = 0;
        } // end loop
      } // end if socket_open
    } // end if remote exists
#endif
```

APPENDIX B. SOURCE CODE PANEL MESSAGE CLASS (MESSAGE.C)

```
// ************************************************************
//   Function to write a PDU socket.  If write fails, display error message,
//     close socket, and terminate program.
// ************************************************************
void Message::writeSocket()
{
    // Set pdu entity address to the npsnet session being controlled
    pdu.vehicleID.address.host = G_data.npsnetAddress.address.host;
    pdu.vehicleID.address.site = G_data.npsnetAddress.address.site;

    if (socket_write((char *) &pdu) != TRUE)
    {
        cerr << "socket failure, write failed" << endl;
        cerr << "Exiting program \n";
        socket_close();

        exit(0);
    }
}


//************************************************************
//   Function to read a PDU socket and distribute the appropriate
//     information throughout the panel display.
//************************************************************
void Message::readSocket()
{

    static GUI_MSG_DATA    *pdu_ptr;
    static struct readstat  rstat;

    if ( (pdu_ptr = socket_read(&rstat)) == NULL )
    {
     // no pdu's available to read
    }
    else
    {
       if ( (pdu_ptr->vehicleID.address.host == G_data.npsnetAddress.address.host)
          && (pdu_ptr->vehicleID.address.site == G_data.npsnetAddress.address.site))
       {
            // update the throttle
            theThrottle->updateThrottle(pdu_ptr);

            // update the instrument component
            theInstrument->updateInstrumentDisplays (pdu_ptr, _pickVehIndex);

            // update the radar screen
            theViewscreen->updateScreen(pdu_ptr);

            // update the tactical map
            theMap->updateMap(pdu_ptr);
         }
      }
}


//***********************************************************************
//   Function to write a PDU socket based on throttle movement.  First set
//     the throttle variable in GUI_MSG_DATA and then write the entire
//     socket structure.
//***********************************************************************
void Message::throttleChanged(int throttle)
{
   pdu.throttleSetting = float(throttle)/100.0;
   writeSocket();
}


//**********************************************************************
//   Function to write a PDU socket based on gun azimuth movement.  First
//     set the azimuth variable in GUI_MSG_DATA and then write the entire
//     socket structure.
//**********************************************************************
void Message::gunAzimuthChanged(int azimuth)
{
   pdu.gunAzimuth = float(azimuth);
   writeSocket();
}


//**********************************************************************
//   Function to write a PDU socket based on gun elevation movement.
//     First set the elevation variable in GUI_MSG_DATA and then write the
//     entire socket structure.
//**********************************************************************
void Message::gunElevationChanged(int elevation)
{
    pdu.gunElevation = float(elevation);
    writeSocket();
}


//************************************************************
// Function to set the vehicle based on user selection.  Function
//   changes the global value of the index of the selected vehicle
//   from the radar screen.  The stealth display will be updated
//   with the new value on the next timer increment within the
//   readSocket function.  No writeSocket is required, since
//   the displayed information does not change the state of NPSNET
//   in any way
//************************************************************
void Message::pickVehicleChanged(int index)
{
    _pickVehIndex = index;
}


//************************************************************
//   Function to write a PDU socket based on joystick movement.
//     First set the values within the GUI_MSG_DATA structure for the
//     joystick inputs, then distribute the socket to NPSNET.
//************************************************************
void Message::joystickChanged(float joystickX, float joystickY)
{
    pdu.joystickX = joystickX;
    pdu.joystickY = joystickY;
    writeSocket();
}


//NEW
//************************************************************
//   Function to write a PDU socket based on FTP position.
//     First set the values within the GUI_MSG_DATA structure for the
//     FTP, then distribute the socket to NPSNET.
//************************************************************
void Message::FTPChanged(int on, float FTPX, float FTPY)
{
   // Set the pdu data fields
   pdu.FTPon = on;
   pdu.FTPXposit = FTPX;
   pdu.FTPYposit = FTPY;

   // Write the pdu to the net
   writeSocket();
}


//*************************************************************
//   Functions to write a PDU socket based on weapon selections.
//     First set the values within the GUI_MSG_DATA structure for the
//     weapon inputs, then distribute the sockets to NPSNET.
//*************************************************************
void Message::targettingState(short state)
{
    if (state)   // targetting mode is on
        pdu.weaponsMode |= GUI_TARGETING;
    else         // targetting mode is off
        pdu.weaponsMode &= !(GUI_TARGETING);

    writeSocket();
}


void Message::primaryWeapon(short state)
{
    if (state)   // mode is on
        pdu.weaponsMode |= GUI_PRIMARY;
    else         // mode is off
        pdu.weaponsMode &= !(GUI_PRIMARY);

    writeSocket();
}


void Message::secondaryWeapon(short state)
{
    if (state)   // mode is on
        pdu.weaponsMode |= GUI_SECONDARY;
    else         // mode is off
        pdu.weaponsMode &= !(GUI_SECONDARY);

    writeSocket();
}


void Message::tertiaryWeapon(short state)
{
    if (state)   // mode is on
       pdu.weaponsMode |= GUI_TERTIARY;
    else         // mode is off
```

```
        pdu.weaponsMode &= !(GUI_TERTIARY);

    writeSocket();
}
```

APPENDIX C. SOURCE CODE PANEL RADAR CLASS (RADAR.H)

```
#ifndef RADAR_H
#define RADAR_H

#include <Vk/VkComponent.h>
#include <Sgm/GlxMDraw.h>

#define MAX_FTP 6
#define MAX_SONOBUOYS 25
#define MAX_SONO_RANGE 1000.0f
#define FTP_CAPTURE_RANGE 500.0f

class Radar : public VkComponent
{

  public:
    Radar(const char *name, Widget parent);
    ~Radar();
    virtual const char *className();

    void updateRadar(float posX, float posY, float hdg);
    void setRange(int);

    //NEW
    void FTPState(short);
    void sonobuoyLaunch(short);

  protected:

    void ginit(GlxDrawCallbackStruct *cb);
    void expose(GlxDrawCallbackStruct *cb);
    void resize(GlxDrawCallbackStruct *cb);
    void inputFTP(GlxDrawCallbackStruct *cb);
    void initialize_gl();

    static void exposeCallback(Widget w, XtPointer clientData, XtPointer callData);
    static void ginitCallback(Widget w, XtPointer clientData, XtPointer callData);
    static void resizeCallback(Widget w, XtPointer clientData, XtPointer callData);
    static void inputCallback(Widget w, XtPointer clientData, XtPointer callData);

    void computeworldcoordinate(int, int, float *, float *);
```

```
        void monitorNet();
        void Update_map();
        int pickVehicle();
        void draw_stuff_on_map(void);
        void drawBoundaries();

        //NEW - FTP/sonobuoy functions
        void drawHeadingBug();
        void drawSonobuoys();
        void drawFTPs();
        void deleteFTP(int);

        //NEW - function to compute the 2D distance between 2 points
        float computeTheDist(float, float, float, float);

    private:

        //NEW - determines whether we are in add mode or delete mode
        short FTP_add_mode;

        //NEW - structure for the FTP's position
        typedef struct
        {
          float x_posit;
          float y_posit;
        } FTP_POSIT;

        //NEW - structure for the FTP information
        typedef struct
        {
          int used;
          FTP_POSIT position;
        } FTP_INFO;

        //NEW - array to hold all FTPs
        FTP_INFO FTP_array[MAX_FTP];

        //NEW - structure for the sonobuoy's position
        typedef struct
        {
          float sono_x_posit;
          float sono_y_posit;
        } SONO_POSIT;

        //NEW - structure for the sonobuoy information
        typedef struct
        {
```

```
        int sono_used;
        SONO_POSIT sono_position;
      } SONO_INFO;

      //NEW - array to hold all sonobuoys
      SONO_INFO Sono_array[MAX_SONOBUOYS];
};

extern Radar *theRadar;
#endif
```

APPENDIX D. SOURCE CODE PANEL RADAR CLASS (RADAR.C)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <stream.h>
#include <iostream.h>

#include “radar.H”
#include “message.H”

#include “entity.H”
#include “netutil.H”
#include “global.h”
#include “fonts/fontdef.h”
#include “constants.h”

// Constants that define the GL state for the GL widget.
static GLXconfig glxConfig [] =
{
   {GLX_NORMAL, GLX_DOUBLE, TRUE},
   {GLX_NORMAL, GLX_RGB, TRUE},
   {GLX_NORMAL, GLX_ZSIZE, GLX_NOCONFIG},
   {0, 0, 0}
};

// set up initial radar center position as center of the world
float centerX = (G_data.maxX - G_data.minX)/2.0;
float centerY = (G_data.maxY - G_data.minY)/2.0;
float centerHdg = 0.0;

static short pickbuffer[MAX_VEH];

//NEW - variables to determine which sonobuoy is next
static int next_Sonobuoy = 0;

float _range = 1.0;
float _posX;
float _posY;
float _hdg;

Radar *theRadar = NULL;
```

```

Radar::Radar(const char *name, Widget parent) : VkComponent(name)
{
    theRadar = this;
    int count;
    Arg args[10];

    count = 0;

    //NEW - initialize ASW mode to off
    FTP_add_mode = 0;

    //NEW - initialize values for the FTP array
    for(int ax = 0; ax < MAX_FTP; ax++)
    {
      FTP_array[ax].used = FALSE;
      FTP_array[ax].position.x_posit = 0.0f;
      FTP_array[ax].position.y_posit = 0.0f;
    }

    //NEW - initialize values for the sonobuoy array
    for(ax = 0; ax < MAX_SONOBUOYS; ax++)
    {
      Sono_array[ax].sono_used = FALSE;
      Sono_array[ax].sono_position.sono_x_posit = 0.0f;
      Sono_array[ax].sono_position.sono_y_posit = 0.0f;
    }

    // replace the default translation table for the GlxNinputCallback
    // with a new one that only responds to mouse button down events,
    // rather than the default of all button and keyboard events.
    String translations = "<BtnDown>:glxInput()";

    XtSetArg(args[count], GlxNglxConfig, glxConfig); count++;

    // use the newly defined translation table to replace to default one
    XtSetArg(args[count], XmNtranslations,  XtParseTranslationTable
              (translations)); count++;

    _baseWidget = GlxCreateMDraw(parent, _name, args, count);

    XtAddCallback(_baseWidget, GlxNexposeCallback, &Radar::exposeCallback,
           (XtPointer) this);
    XtAddCallback(_baseWidget, GlxNginitCallback, &Radar::ginitCallback,
           (XtPointer) this);
    XtAddCallback(_baseWidget, GlxNresizeCallback, &Radar::resizeCallback,
           (XtPointer) this);
```

```
   XtAddCallback(_baseWidget, GlxNinputCallback, &Radar::inputCallback,
         (XtPointer) this);
}

void Radar::exposeCallback(Widget, XtPointer, XtPointer callData)
{
   GlxDrawCallbackStruct *cb = (GlxDrawCallbackStruct *) callData;

   GLXwinset(XtDisplay(theRadar->baseWidget()),XtWindow(theRadar->
                       baseWidget()));
   theRadar->expose(cb);
}

void Radar::ginitCallback(Widget, XtPointer, XtPointer callData)
{
   GlxDrawCallbackStruct *cb = (GlxDrawCallbackStruct *) callData;

   GLXwinset(XtDisplay(theRadar->baseWidget()), XtWindow(theRadar->
                       baseWidget()));
   theRadar->ginit(cb);
}

void Radar::resizeCallback(Widget, XtPointer, XtPointer callData)
{
   GlxDrawCallbackStruct *cb = (GlxDrawCallbackStruct *) callData;

   GLXwinset(XtDisplay(theRadar->baseWidget()), XtWindow(theRadar->
                       baseWidget()));
   theRadar->resize(cb);
}

void Radar::inputCallback(Widget, XtPointer, XtPointer callData)
{
   //NEW - The panel is in ASW vehicle mode and new FTP's are to be added,
   // so make the inputFTP callback
   GlxDrawCallbackStruct *cb = (GlxDrawCallbackStruct *) callData;
   GLXwinset(XtDisplay(theRadar->baseWidget()), XtWindow(theRadar->
                       baseWidget()));
   theRadar->inputFTP(cb);

   // We can't add FTP's, so change the pick vehicle for stealth
   theMessage->pickVehicleChanged ( theRadar->pickVehicle() );

}


//*************************************************************
```

```
void Radar::ginit(GlxDrawCallbackStruct *)
{
   setRange(100);
   initialize_gl();

   // wait until the gl window is initialized to start drawing to
   // it within the timer loop, otherwise ...core dump...
   theMessage->_timer->start();

   // start listening to the net for entities to display
   theRadar->monitorNet();
}

Radar::~Radar()
{
   // Empty
}

const char *Radar::className()
{
   return "Radar";
}

//************************************************************
void Radar::expose(GlxDrawCallbackStruct *)
{
   updateRadar(G_data.world_x_size/2.0, G_data.world_y_size/2.0, 0.0);
}

//************************************************************
void Radar::resize(GlxDrawCallbackStruct *cb)
{
   viewport(0, (Screencoord) cb->width-1,
                  0, (Screencoord) cb->height-1);
   updateRadar(G_data.world_x_size/2.0, G_data.world_y_size/2.0, 0.0);
}

//NEW - this designates the location for a new FTP
//************************************************************
void Radar::inputFTP(GlxDrawCallbackStruct *cb)
{
  // Variables for the location of the FTP in world coordinates
  float wx, wy;

  // Variable for which FTP is the next to be input
  int next_FTP = 0;

```

```
// Find out which FTP is the next to be assigned
while ((next_FTP < MAX_FTP) && (FTP_array[next_FTP].used == TRUE))
  next_FTP++;

if (theRadar->FTP_add_mode)
{
  if (next_FTP != MAX_FTP)
  { // one is available, so add it; otherwise do nothing

    // Call computeworldcoordinate() to convert the screen coordinate
    //  on the radar screen to terrain coordinates
    computeworldcoordinate((Screencoord)cb->event->xbutton.x,
                                    (Screencoord)cb->event->xbutton.y, &wx, &wy);

    // Set the fields of the FTP array
    FTP_array[next_FTP].used = TRUE;
    FTP_array[next_FTP].position.x_posit = wx;
    FTP_array[next_FTP].position.y_posit = wy;

    // the top priority FTP changed, so send message to NPSNET
    if (next_FTP == 0)
      theMessage -> FTPChanged(FTP_array[next_FTP].used,
                                          FTP_array[next_FTP].position.x_posit,
                                          FTP_array[next_FTP].position.y_posit);

  } // end if next_FTP != MAX_FTP

} // end if FTP_add_mode on

else // FTP add mode is off, so let the user pick an FTP and delete it
{
  // Call computeworldcoordinate() to convert the screen coordinate
  //  on the radar screen to terrain coordinates
  computeworldcoordinate((Screencoord)cb->event->xbutton.x,
                                  (Screencoord)cb->event->xbutton.y, &wx, &wy);

  for(int ix=0;ix<MAX_FTP;ix++)
  {
    // If the FTP got picked, delete it
    if ((( wx >= (FTP_array[ix].position.x_posit - (_range/20.0f))) &&
         ( wx <= (FTP_array[ix].position.x_posit + (_range/20.0f)))) &&
        (( wy >= (FTP_array[ix].position.y_posit - (_range/20.0f))) &&
         ( wy <= (FTP_array[ix].position.y_posit + (_range/20.0f)))))
      deleteFTP(ix);
  } // end try to delete an FTP

} // end else state

} // end inputFTP()

//************************************************************
void Radar::initialize_gl()
{
    float scrnaspect;            // aspect ratio value
    long xscrnsize;              // size of screen in x used to set globals

    xscrnsize = getgdesc(GD_XPMAX);          // get/set screen size[/aspect]
    if (xscrnsize == 1280)
    {
        keepaspect(5, 4);
        scrnaspect = 1.25;
    }
    else if (xscrnsize == 1023)
    {
        keepaspect(4, 3);
        scrnaspect = 1.34;
    }
    else
    {
        fprintf(stderr, “Something’s EXTREMELY wrong:  “);
        fprintf(stderr, “xscrnsize=%d\n”, xscrnsize);
        exit(-1) ;
    }

    // Always turn subpixel true.
    subpixel(TRUE);

    //define the font
    fontdef(1L, G_data.hud_font_file);

    doublebuffer();
}

//************************************************************
void Radar::monitorNet()
{
    G_data.doing = MONITOR;

    //there will be no outgoing PDUs
    G_data.net = RECEIVE;

    //stop generating PDUs, if we are generating them
    for(int ix =0;ix<MAX_VEH;ix++)
    {
        if (G_vehlist[ix].vehptr != NULL)
        {   //is it an active entity
            if(G_vehlist[ix].vehptr->get_C2_flag())
            {
                //we have to deactivate the vehicle
                deactiveate__entity(ix);
            }
        }
    }
}

//***************************************************************
// taken from the file inst_ptr.c to draw the artificial horizon
void Radar::updateRadar(float posX, float posY, float hdg)
{
    //set the global position x and y based on the socket input
    _posX = posX;
    _posY = posY;
    _hdg = hdg;

    // set the correct window to draw the radar into
    GLXwinset(XtDisplay(theRadar->baseWidget()), XtWindow(theRadar->
                        baseWidget()));

    // set the background color for the radar screen
    RGBcolor(0, 0, 0);
    clear();

    // display only the selected range centered at posX and posY
    ortho2(_posX - _range, _posX + _range,
                _posY - _range, _posY + _range);

    // translate (-_posX, -_posY, 0) to place origin at current position
    translate (_posX, _posY, 0.0);

    //NEW - real radar screens don’t rotate, so don’t rotate (heading);
//    rot(_hdg, ‘z’);

    // translate (posX, posY, 0) to return view back to normal
    translate (-_posX, -_posY, 0.0);

    // draw radar range rings
    linewidth(1);
    cpack (0x00ff0000);
    circ(posX,posY,20000.0f);
    cpack (0x0000ff00);
    circ(posX,posY,10000.0f);
    cpack (0x0000ffff);
    circ(posX,posY,5000.0f);
    cpack (0x000000ff);
    circ(posX,posY,1000.0f);
    linewidth(1);

    drawBoundaries();
    Update_map();

} // updateRadar(float,float,float)

//*************************************************************
void Radar::Update_map()
{
    static int count = 0;
    static double oldtime =0.0;

    //if the network is open, update the time
    if(G_data.net != CLOSED)
    {

        G_data.sim_time = pfGetTime();
        if(oldtime <  G_data.sim_time)
            G_data.delta_time = G_data.sim_time - oldtime;
        else     // time reset
            G_data.delta_time = 0;
        oldtime = G_data.sim_time;

        //move all of the icons
        update_entities();

        if(G_data.net != OPEN)
        {
            //where are all of the entities
            draw_stuff_on_map();
        }
    }

    if(G_data.net != CLOSED)
    {
        parse_net_pdus();
    }

    swapbuffers();
    count ++;
}

```

```

//************************************************************
void Radar::draw_stuff_on_map()
// draws the new and improved 2D map on the screen
{
    int ix, inttype;
    float pos[3];
    char text[40];

    //use the pretty pictures
    font(1);

    for(ix=0;ix<MAX_VEH;ix++)
    {
        if(G_vehlist[ix].vehptr == NULL) continue;
        pfCopyVec3(pos,G_vehlist[ix].vehptr->getposition());
        inttype = G_vehlist[ix].type;
        if (inttype < 1 )
           sprintf(text,”%c”,’?’);
        else if ( (inttype >= 1) && (inttype <= 5) )
           sprintf(text,”%c”,’J’);
        else if ( (inttype >= 6) && (inttype <= 10) )
           sprintf(text,”%c”,’E’);
        else if ( (inttype >= 11) && (inttype <= 20) )
           sprintf(text,”%c”,’Q’);
        else if ( (inttype >= 20) && (inttype <= 25) )
           sprintf(text,”%c”,’R’);
        else if ( (inttype >= 26) && (inttype <= 30) )
           sprintf(text,”%c”,’C’);
        else if ( (inttype >= 31) && (inttype <= 35) )
           sprintf(text,”%c”,’?’);
        else if ( (inttype >= 36) && (inttype <= 40) )
           sprintf(text,”%c”,’A’);
        else if ( (inttype >= 41) && (inttype <= 45) )
           sprintf(text,”%c”,’I’);
        else if ( (inttype >= 46) && (inttype <= 50) )
           sprintf(text,”%c”,’I’);
        else if ( (inttype >= 51) && (inttype <= 55) )
           sprintf(text,”%c”,’G’);
        else if ( (inttype >= 56) && (inttype <= 57) )
           sprintf(text,”%c”,’A’);
        else if ( (inttype >= 58) && (inttype <= 60) )
           sprintf(text,”%c”,’I’);
        else if ( (inttype >= 61) && (inttype <= 70) )
           sprintf(text,”%c”,’B’);
//      DON’T PRINT SUBS ON DISPLAY
        else if ( (inttype >= 71) && (inttype <= 75) )
```

```
            sprintf(text,”%c”,’ ‘);
          else if ( (inttype >= 76) && (inttype <= 80) )
            sprintf(text,”%c”,’?’);
          else if ( (inttype >= 81) && (inttype <= 95) )
            sprintf(text,”%c”,’&’);
          else if ( (inttype >= 96) && (inttype <= 99) )
            sprintf(text,”%c”,’?’);
          else if ( (inttype >= 100) && (inttype <= 199) )
            sprintf(text,”%c”,’/’);
          else
            sprintf(text,”%c”,’x’);


          switch(G_vehlist[ix].vehptr->getforce())
          {
            case ForceID_Blue:
              cpack(0xffff8f00);
              break;
            case ForceID_Red:
              cpack(0xff0000ff);
              break;
            case ForceID_White:
              cpack(0xffffffff);
              break;
            default:
              cpack(0xff00ff00);
              break;
          }

           cmov2(pos[X],pos[Y]);
           charstr(text);
      }

      //back to the normal text
      font(0);

      //NEW - this calls the function to draw the heading bug
      drawHeadingBug();

      //NEW - draw the sonobuoys and their bearing lines
      drawSonobuoys();

      //NEW - draw the FTPs
      drawFTPs();
  }

  //**************************************************************
```

```
int Radar::pickVehicle()
// draws pick map on the radar screen to evaluate mouse input
{
    static int ix;
    static float pos[3];
    static char text[40];
    static int numofhits;

    // set the correct window to draw the pick map into
    GLXwinset(XtDisplay(theRadar->baseWidget()), XtWindow(theRadar->
                        baseWidget()));

    //set up the pick buffers
    pushmatrix();
    pick(pickbuffer,MAX_VEH);

    // **************** Set up the window *****************
    ortho2(_posX - _range, _posX + _range,
                _posY - _range, _posY + _range);

    // translate (-_posX, -_posY, 0) to place origin at current position
    translate (_posX, _posY, 0.0);

    // rotate (heading);
    rot(_hdg, ‘z’);

    // translate (posX, posY, 0) to return view back to normal
    translate (-_posX, -_posY, 0.0);

    font(1);

    for(ix=0;ix<MAX_VEH;ix++)
    {
        if(pos,G_vehlist[ix].vehptr == NULL) continue;
        pfCopyVec3(pos,G_vehlist[ix].vehptr->getposition());

        //store the location
        loadname((short)ix);

        //figure out where the guy is
        sprintf(text,”%c”,’E’);
        cpack(0xffffffff);
        cmov2(pos[X],pos[Y]);
        charstr(text);
    }
    font(0);

    //number of things picked from the screen
    numofhits = (int)endpick(pickbuffer);

    if(numofhits > 0)
    {
        popmatrix();
        return (pickbuffer[1]);
    }
    else // no vehicle was selected
    {
        popmatrix();
        return (0);
    }
}

//NEW - changed from old code
//*************************************************************
// computeworldcoordinate -
//   This function computes the world coordinate corresponding to the
//   input mouse coordinate.
//   The input mouse coordinate is in the X window system so the
//   origin is in the upper left corner.
//
//   Inputs:
//      int x,y = input mouse coordinate in X coordinate system.
//
//   Outputs:
//      float *wx, *wy = returned world coordinate.
//

void Radar::computeworldcoordinate(int x, int y, float *wx, float *wy)
{
   // Find the world x coordinate
   *wx = (( 2 * _range )/324.0f) * (x) + ( _posX - _range );

   // Find the world y coordinate
   *wy = (( 2 * _range )/324.0f) * ((324.0f - 1) - y) +  ( _posY - _range );
}

//*************************************************************
void Radar::setRange(int range)
{
    //set range as a function of the % of the world size
    //NEW - only show the shortest distance that can be seen
    if (G_data.maxX < G_data.maxY)
        _range = ((G_data.maxX * float(range) / 100.0) / 2.0);
    else
```

```
          _range = ((G_data.maxY * float(range) / 100.0) / 2.0);

    if (_range < 1.0)
          _range = 1.0;
}

//************************************************************
// draw the boundaries of the world
void Radar::drawBoundaries()
{
    static float p[4][2];
    p[0][0] = G_data.minX;
    p[0][1] = G_data.minY;

    p[1][0] = G_data.minX;
    p[1][1] = G_data.maxY;

    p[2][0] = G_data.maxX;
    p[2][1] = G_data.maxY;

    p[3][0] = G_data.maxX;
    p[3][1] = G_data.minY;

    cpack(0xFF888888);

    bgnclosedline();
          v2f(p[0]);
          v2f(p[1]);
          v2f(p[2]);
          v2f(p[3]);
    endclosedline();
}

//NEW - all FTP functionality
// This function turns on/off the pick functions for FTPs
void Radar::FTPState(short state)
{
   // Allow the user to input a new a FTP
   if (state)
      theRadar->FTP_add_mode = 1;

   // Allow the user to pick a FTP and delete it
   else
      theRadar->FTP_add_mode = 0;
}


```

```
//NEW - Draw the heading bug
void Radar::drawHeadingBug()
{
  // 2x2 array for starting point and endpoint for the heading bug
  float head_vec[2][2];

  // This is the starting point, the center of the display
  head_vec[0][0] = _posX;
  head_vec[0][1] = _posY;

  // The endpoint is in the direction of heading, scaled so that
  //  it appears the same length no matter what the range setting of
  //  the radar is.
  head_vec[1][0] = _posX + ((_range/5.0f) * sinf(_hdg * DEG_TO_RAD));
  head_vec[1][1] = _posY + ((_range/5.0f) * cosf(_hdg * DEG_TO_RAD));

  // draw the heading bug
  linewidth(1);
  cpack (0x00ffffff); // color white
  bgnline();
    v2f(head_vec[0]);
    v2f(head_vec[1]);
  endline();
}


//NEW - this function adds a new sonobuoy to the sonobuoy array
void Radar::sonobuoyLaunch(short state)
{
  // State is true so add a new sonobuoy; otherwise, do nothing
  if ((state) && (next_Sonobuoy < MAX_SONOBUOYS))
  {
    Sono_array[next_Sonobuoy].sono_used = TRUE;
    Sono_array[next_Sonobuoy].sono_position.sono_x_posit = _posX;
    Sono_array[next_Sonobuoy].sono_position.sono_y_posit = _posY;
    next_Sonobuoy++;
  }
}


//NEW - this draws the sonobuoys and lines of bearing to the sub
void Radar::drawSonobuoys()
{
  // This if for the font to be used
  char text[40];

  cpack(0x0000ff00); // color green
```

```

// Draw the sonobuoys and their bearing lines
for(int ax = 0; ax < MAX_SONOBUOYS; ax++)
{
  // Variables for distance calculation
  float dist_to_contact = MAX_SONO_RANGE;
  float temp_distance = 0.0f;

  // Variable for which ID we are going to target
  int contact_ID = -1;

  // Array to hold the endpoints of bearing lines
  float bearing_line[2][2] = {0};

  // Arrays for target position
  float target_pos[3], temp_pos[3];

  if (Sono_array[ax].sono_used == TRUE)
  {
    // Sonobuoy is the first point of the bearing line
    bearing_line[0][0] = Sono_array[ax].sono_position.sono_x_posit;
    bearing_line[0][1] = Sono_array[ax].sono_position.sono_y_posit;

    // First draw the sonobuoy symbol
    font(1);
    sprintf(text,"%c",'0');
    cmov2(bearing_line[0][0], bearing_line[0][1]);
    charstr(text);

    // Now draw the bearing line to the target
    for(int bx = 0; bx < MAX_VEH; bx++)
    {
      if ((G_vehlist[bx].vehptr != NULL) &&
          (G_vehlist[bx].vehptr->gettype() == SUBMERSIBLE))
      { // this is a SUB, so find if it is the closest
        pfCopyVec3(temp_pos, G_vehlist[bx].vehptr->getposition());
        temp_distance = computeTheDist(bearing_line[0][0],
                                       bearing_line[0][1],
                                       temp_pos[X], temp_pos[Y]);

        // shorter distance test
        if (temp_distance < dist_to_contact)
        { // it is closer, so target it
          pfCopyVec3(target_pos, temp_pos);
          contact_ID = bx;
          dist_to_contact = temp_distance;

```

```
            } // end if closer

          } // end if this is a sub

        } // end search for closest SUB to draw to

        if (contact_ID != -1)
        { // we have the closest contact, so draw the bearing line
          // Target is the endpoint of the bearing line
          bearing_line[1][0] = target_pos[X];
          bearing_line[1][1] = target_pos[Y];

          // Draw the bearing line
          font(0);
          linewidth(1);
          bgnline();
            v2f(bearing_line[0]);
            v2f(bearing_line[1]);
          endline();

        } // end if we have a contact

      } // end if the sonobuoy is to be drawn

    } // end for all possible sonobuoys

} // end drawSonobuoys()


//NEW - this draws the sonobuoys and lines of bearing to the sub
void Radar::drawFTPs()
{
  // This if for the font to be used
  char text[40];

  // First check to see if the vehicle is close enough to
  //   capture the #1 FTP; is so, delete and launch a sonobuoy
  if ((FTP_array[0].used == TRUE) &&
        ((computeTheDist(_posX, _posY,
                                   FTP_array[0].position.x_posit,
                                   FTP_array[0].position.y_posit))
             <= FTP_CAPTURE_RANGE ))
      {
        deleteFTP(0);
        sonobuoyLaunch(1);
      } // end capture FTP

```

```
   // Draw the FTPs
   font(1);
   cpack(0x00ff00ff); // color cyan
   for(int cx = 0; cx < MAX_FTP; cx++)
   {
     // Only put it on the screen if it is active
     if (FTP_array[cx].used == TRUE)
     {
       // Designate the FTP font to be used in order of priority
       if (cx == 0)
         sprintf(text,”%c”,’4’);
       else if (cx == 1)
         sprintf(text,”%c”,’5’);
       else if (cx == 2)
         sprintf(text,”%c”,’6’);
       else if (cx == 3)
         sprintf(text,”%c”,’7’);
       else if (cx == 4)
         sprintf(text,”%c”,’8’);
       else
         sprintf(text,”%c”,’9’);

       // Put the character in the correct radar position
       cmov2(FTP_array[cx].position.x_posit,
                 FTP_array[cx].position.y_posit);
       charstr(text);
     } // end if FTP is used

   } // end for all possible sonobuoys
   font(0);

} // end drawFTPs()


//NEW - this deletes the selected FTP and changes the priority
//  of those that remain
void Radar::deleteFTP(int FTPid)
{
   // The number deleted is the top priority FTP, so send
   //  new message to NPSNET
   if ((FTPid == 0) && (FTP_array[FTPid].used == TRUE))
     theMessage -> FTPChanged(FTP_array[FTPid + 1].used,
                                       FTP_array[FTPid + 1].position.x_posit,
                                       FTP_array[FTPid + 1].position.y_posit);

   // Rearrange the rest of the array
   while (FTPid < (MAX_FTP - 1))
```

```
    {
       FTP_array[FTPid] = FTP_array[FTPid + 1];
       FTPid++;
    } // end rearrange FTPs

    // Set the last FTP to unused
    FTP_array[FTPid].used = FALSE;

} // end deleteFTP()


//NEW - used to calculate the 2D distance between 2 points using the
// Pythagorean theorem
float Radar::computeTheDist(float from_x, float from_y,
                                        float to_x, float to_y)
{
    return(sqrt(((to_y - from_y) * (to_y - from_y)) +
                     ((to_x - from_x) * (to_x - from_x))));
}
```

LIST OF REFERENCES

[HART94] Hartman, J., Creek, P., “IRIS Performer Programming Guide”, Document Number 007-1680-020, Silicon Graphics, Inc., Mountain View, California, 1994.

[LWSM92] Commander, Naval Air Systems Command, “LAMPS MK III Weapon System Manual”, NAVAIR A1-H60BB-NFM-010, Washington, D.C., 1 March 1992.

[MCMA95] McMahan, C. B., “NPSNET IV: An Object-Oriented Interface for a Three-Dimensional Virtual World”, Master’s Thesis, Naval Postgraduate School, Monterey, California, December, 1994.

[NTPS90] Commander, Naval Air Systems Command, “NATOPS Flight Manual, Navy Model SH-60B Aircraft”, NAVAIR A1-H60BB-NFM-000, Washington, D.C., 15 June 1990.

[PROU86] Prouty, Raymond W., *Helicopter Performance, Stability, and Control*, PWS Publishers, Boston, Massachusetts, 1986.

[ZYDA94] Zyda, Michael J., Macedonia, Michael R., Pratt, David R., Barham, Paul T., Zezwitz, Steven, “NPSNET: A Network Software Architecture for Large Scale Virtual Environments”, Naval Postgraduate School, Monterey, California, 1994.

[ZYDA95] Zyda, Michael J., Macedonia, Michael R., Brutzman, Donald P., Pratt, David R., Barham, Paul T., “NPSNET: A Multi-Player 3D Virtual Environment Over the Internet”, *Proceedings of 1995 Symposium on Interactive 3D Graphics*, 1995.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   Cameron Station
   Alexandria, VA 22304-6145

2. Dudley Knox Library 2
   Code 013
   Naval Postgraduate School
   Monterey, CA 93943-5101

3. Dr. Ted Lewis, Chairman and Professor. 1
   Computer Science Department Code CS/LT
   Naval Postgraduate School
   Monterey, CA 93943-5000

4. Dr. David R. Pratt, Assistant Professor. 3
   Computer Science Department Code CS/PR
   Naval Postgraduate School
   Monterey, CA 93943-5000

5. John S. Falby, Lecturer 3
   Computer Science Department Code CS/FJ
   Naval Postgraduate School
   Monterey, CA 93943-5000

6. Dr. Michael J. Zyda, Professor 10
   Computer Science Department Code CS/ZK
   Naval Postgraduate School
   Monterey, CA 93943-5000

7. Paul Barham, Computer Specialist 1
   Computer Science Department Code CS/Barham
   Naval Postgraduate School
   Monterey, CA 93943-5000

8. Don Selvy 1
   A&T, Inc.
   1215 Jefferson Davis Highway, Suite 310
   Arlington, VA 22202-4302

9. Lt Frederick C. Lentz, III 2
818 Sherri St.
Ridgecrest, CA   93555-3436